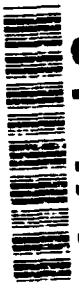
IDA PAPER P-2422

ASSESSMENTS OF SELECTED REAL-TIME COMPUTING TECHNOLOGIES

Karen D. Gordon
Kevin J. Rappoport

Dennis W. Fife, *Task Leader*

July 1991



Prepared for
Strategic Defense Initiative Organization

INSTITUTE FOR DEFENSE ANALYSES
1801 N. Beauregard Street, Alexandria, Virginia 22311-1772

IDA Log No. HQ 90-035650

DEFINITIONS

IDA publishes the following documents to report the results of its work.

Reports

Reports are the most authoritative and most carefully considered products IDA publishes. They normally embody results of major projects which (a) have a direct bearing on decisions affecting major programs, (b) address issues of significant concern to the Executive Branch, the Congress and/or the public, or (c) address issues that have significant economic implications. IDA Reports are reviewed by outside panels of experts to ensure their high quality and relevance to the problems studied, and they are released by the President of IDA.

Group Reports

Group Reports record the findings and results of IDA established working groups and panels composed of senior individuals addressing major issues which otherwise would be the subject of an IDA Report. IDA Group Reports are reviewed by the senior individuals responsible for the project and others as selected by IDA to ensure their high quality and relevance to the problems studied, and are released by the President of IDA.

Papers

Papers, also authoritative and carefully considered products of IDA, address studies that are narrower in scope than those covered in Reports. IDA Papers are reviewed to ensure that they meet the high standards expected of refereed papers in professional journals or formal Agency reports.

Documents

IDA Documents are used for the convenience of the sponsors or the analysts (a) to record substantive work done in quick reaction studies, (b) to record the proceedings of conferences and meetings, (c) to make available preliminary and tentative results of analyses, (d) to record data developed in the course of an investigation, or (e) to forward information that is essentially unanalyzed and unevaluated. The review of IDA Documents is suited to their content and intended use.

The work reported in this document was conducted under contract MDA 903 89 C 0003 for the Department of Defense. The publication of this IDA document does not indicate endorsement by the Department of Defense, nor should the contents be construed as reflecting the official position of that Agency.

This Paper has been reviewed by IDA to assure that it meets high standards of thoroughness, objectivity, and appropriate analytical methodology and that the results, conclusions and recommendations are properly supported by the material presented.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	July 1991	Final

4. TITLE AND SUBTITLE
Assessments of Selected Real-Time Computing Technologies

5. FUNDING NUMBERS
MDA 903 89 C 0003

Task T-R2-597.2

6. AUTHOR(S)
Karen D. Gordon, Kevin J. Rappoport

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Institute for Defense Analyses (IDA)
1801 N. Beauregard St.
Alexandria, VA 22311-1772

8. PERFORMING ORGANIZATION REPORT NUMBER
IDA Paper P-2422

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
SDIO/ENA
Room 1E149, The Pentagon
Washington. D.C. 20301-7100

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release, unlimited distribution: 22 October 1991.

12b. DISTRIBUTION CODE
2A

13. ABSTRACT (Maximum 200 words)

Developing dependable software for large, complex, real-time systems is one of the major challenges now facing the software industry. The software R&D community is responding to this challenge; numerous efforts have been initiated on various aspects of real-time software development. In this paper, we review and evaluate ongoing R&D efforts in light of the needs of strategic defense systems. We identify and discuss four recent developments that hold promise for facilitating the design and implementation of real-time software for strategic defense systems: (1) rate monotonic scheduling theory, (2) real-time extensions to the IEEE Portable Operating System Interface for Computer Environments (POSIX), (3) several distributed real-time operating system prototypes, and (4) various methods for enhancing real-time system robustness by trading precision of results for timeliness of results. We also point out an area of major concern to real-time software developers and, in particular, to the SDIO: the lack of analytical methods for evaluating the performance of complex real-time systems. We conclude with a series of recommendations on how the SDIO should follow up on the real-time R&D topics covered in the paper.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Real-time Operating Systems; Portable Operating System Interface for Computer Environments (POSIX); Rate Monotonic Scheduling Theory.	126
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	SAR

IDA PAPER P-2422

ASSESSMENTS OF SELECTED REAL-TIME COMPUTING TECHNOLOGIES

Karen D. Gordon
Kevin J. Rappoport

Dennis W. Fife, *Task Leader*

July 1991

INSTITUTE FOR DEFENSE ANALYSES

Contract MDA 903 89 C 0003
Task T-R2-597.2

PREFACE

This paper was prepared for the Strategic Defense Initiative Organization (SDIO) in response to the requirement of Subtask Order T-R2-597.2, "SDIO Software Technology Plan," to review and evaluate current research related to developing software for real-time systems. Its purpose is twofold: first, to identify research efforts that hold promise for facilitating the development of real-time software for strategic defense systems; and second, to identify areas of concern that demand further research.

This paper was reviewed by the following members of the Institute for Defense Analyses management and staff: Dr. Richard L. Wexelblat, Mr. W. T. Mayfield, Mr. Stephen H. Edwards, Dr. Reginald N. Meeson, and Dr. James P. Pennell. The authors thank the reviewers for their feedback on earlier drafts of this paper.

The authors of this paper also gratefully acknowledge the contributions of Mr. James Baldo of the Institute for Defense Analyses, Dr. Lawrence W. Dowdy of Vanderbilt University, Mr. William M. Corwin of Intel Corporation, and Dr. C. Douglass Locke of IBM to published articles that are included as appendices of this paper.

TABLE OF CONTENTS

1. INTRODUCTION

The work reported herein was undertaken at the request of the Strategic Defense Initiative Organization (SDIO). It represents a follow-up to the distributed operating system technology assessment documented in the Institute for Defense Analyses (IDA) paper *Strategic Defense System Distributed Operating System R&D: Review and Recommendations* [Gordon and Linn 89]. During the course of conducting the distributed operating system technology assessment, we concluded that real-time computing presents challenges not only to the distributed operating systems that may be utilized in strategic defense systems, but also to strategic defense system software in general.

1.1 BACKGROUND

The demands of real-time computing complicate each phase of the software life cycle. The complicating factor is *time*. Real-time computing systems are distinguished by the significance of the role that time and, in particular, timing constraints play in them. Timing constraints are imposed by the environment in which the real-time computing system exists. Typically, the environment consists of a larger *controlled* system (e.g., an automobile, aircraft, ship, submarine, missile, hospital patient monitoring system, air traffic control system, factory floor, nuclear power plant, etc.), as well as the physical environment in which the controlled system exists. The real-time computing system is the *controlling* element of the larger system. Failure to meet environment-imposed timing constraints can have catastrophic consequences, such as loss of life, loss of the controlled system, or failure of the mission of the controlled system.

In the premier issue of *Real-Time Systems*, the introductory editorial characterizes real-time systems as that category of systems in which "the correctness of the system depends not only on the logical results of computations but also on the time at which the results are produced" [Stankovic 89, 6]. Thus, in real-time systems, timeliness is a first-order system concern. Timing constraints must be taken into account explicitly throughout the software development process. First, timing constraints must be identified. Then, during the design and implementation phases of the life cycle, the timing constraints must be addressed as *real* requirements—as real as the more traditional functional requirements. Likewise, the testing phase must be designed to assure timing correctness as well as functional correctness. Furthermore, whenever modifications are made to a system, during either development or maintenance phases, the impact of the modifications on

timing correctness must be considered.

Timing constraints must be taken into account by real-time systems themselves during the operational phase of their life cycles. Their resource management can be neither fairness-driven nor efficiency-driven, as in interactive timesharing systems. Instead, their resource management must be driven by the time constraints of the *mission* that the real-time system is intended to perform. The timing constraints can be conveyed to the system either explicitly through mechanisms such as deadlines, or implicitly, through mechanisms such as priorities. Moreover, if a timing constraint such as a deadline is missed or is in danger of being missed, it is helpful for the system to be able to recognize the situation and invoke appropriate fault tolerance mechanisms.

The recent proliferation of real-time systems has increased interest in real-time computing. Stankovic of the University of Massachusetts brought the issues of real-time computing to the forefront through publication of his article, "Misconceptions About Real-Time Computing: A Serious Problem for Next-Generation Systems" [Stankovic 88]. It was, in fact, this article that prompted the SDIO to request IDA to further investigate real-time computing technology. In the article, Stankovic defines real-time computing, articulates some of its fundamental issues, and calls upon researchers, developers, and funding agencies to work together in establishing a scientific foundation for real-time computing.

1.2 PURPOSE

The purpose of this paper is two-fold: *first*, to identify promising developments that should be exploited by the SDIO, and *second*, to identify areas of concern that should be addressed by the SDIO through increased research and development (R&D).

1.3 SCOPE

In keeping with its purpose, the scope of this paper is not focused *a priori* on a specific issue in real-time computing. Instead, the scope is, in effect, defined by the activities of the real-time computing community on one hand, and by the real-time computing requirements of strategic defense systems on the other hand.

However, certain specific topics are purposely excluded from the scope of this investigation, since they are the subjects of other investigations being conducted by IDA for the SDIO. These topics are (1) software engineering tools [Fife 87; Wheeler 90] and (2) software testing and evaluation [Brykczynski 90; Youngblut 89] .

1.4 OUTLINE

Sections 2 through 5 of this paper report on four recent developments that hold promise for meeting some of the challenging real-time computing requirements of strategic defense systems:

- Rate monotonic scheduling theory
- IEEE real-time extensions to POSIX (Portable Operating System Interface for Computer Environments)
- Distributed real-time operating system R&D
- Methods for enhancing real-time system robustness by trading precision for timeliness

Section 6 identifies an area of major concern to real-time system developers and, in particular, to the SDIO:

- Lack of analytical methods for evaluating the performance of aperiodic real-time systems.

Section 7 concludes the paper with a series of recommendations on how the SDIO should follow up on each of the topics highlighted in this paper.

2. RATE MONOTONIC SCHEDULING THEORY

In general, real-time systems have been built through *ad hoc* techniques, often at inordinate expense. As systems become ever larger and more complex, a more disciplined approach becomes essential. Rate monotonic scheduling theory offers a disciplined approach for certain real-time systems—those that those can be developed in a periodic framework.

The basic rate monotonic algorithm is an algorithm for scheduling periodic tasks with hard deadlines. It dates back to 1973, when it was published by Liu and Layland [Liu and Layland 73]. In recent years, it has become the basis of a whole body of scheduling theory, developed in large part by researchers at Carnegie Mellon University and the Software Engineering Institute (SEI). The evolving theory has made significant progress toward accommodating aperiodic tasks, transient overload, and task synchronization.

The rate monotonic algorithm and the theory that has evolved around it offer several important features: straightforward assignment of scheduling attributes, straightforward scheduler, low overhead, efficiency, predictability, adaptability, applicability to Ada real-time systems, and extensibility. These features, which are elaborated in the following section, combine to make the rate monotonic algorithm the scheduling algorithm of choice for periodic-based real-time systems.

Synopsis of the Rate Monotonic Scheduling Algorithm

Let $\tau_1, \ldots, \tau_m$ be a set of m periodic tasks, with task τ_i having a period (or constant interarrival time) of T_i and computation requirements of C_i, $i = 1, \ldots, m$. Each arrival (i.e., occurrence) of task τ_i initiates a period of length T_i time units, during which it must receive C_i time units of computation. The end of the period represents the "hard deadline" of the arrival that initiated the period. It also coincides with the occurrence of the next arrival, which marks the beginning of the next period. Since task τ_i requires C_i time units of computation every T_i time units, its utilization of the processor is C_i/T_i. [1]

The rate monotonic algorithm is an optimal preemptive, static-priority-driven uniprocessor scheduling algorithm for periodic tasks with hard deadlines as defined in the

1. The notation used here is based on the notation in [Liu and Layland 73].

previous paragraph.[2] As a *preemptive, priority-driven* algorithm, it ensures that the processor is always executing the highest priority task in the processor's ready queue (unless of course the queue is empty, in which case the processor is idle). If a task of higher priority than the currently executing task joins the queue, then the currently executing task is preempted, and the processor begins executing the newly arrived (higher priority) task.

As a *static* algorithm, the rate monotonic algorithm assigns priorities that are determined prior to execution and remain fixed over time. In particular, it assigns static priorities to tasks according to the lengths of their periods; tasks with shorter periods are assigned higher priorities. Alternatively stated, tasks with higher arrival rates are assigned higher priorities, making task priority a monotonically increasing function of task arrival rate; hence the term "rate monotonic."

The rate monotonic algorithm was shown to be optimal within the class of preemptive, static-priority-driven scheduling algorithms by Liu and Layland [Liu and Layland 73]. This means that no other algorithm of this class can schedule a set of periodic tasks that cannot also be scheduled by the rate monotonic algorithm. In addition to establishing optimality, Liu and Layland established a *sufficient* condition for the schedulability of an arbitrary task set. The condition is that the total processor utilization U of the task set is no more than $ln\ 2$, i.e.,

$$U = \sum_{i=1}^{i=m} (C_i/T_i) \leq ln\ 2\ (\approx 0.693)$$

Thus, for any task set of any size whose total processor utilization is less than or equal to $ln\ 2$, the rate monotonic algorithm guarantees that all deadlines of the task set will be met. Under more restrictive conditions, the total utilization can be higher.

The rate monotonic algorithm offers several important features:

a. Straightforward assignment of scheduling attributes: For the rate monotonic algorithm, the scheduling attributes are priorities. Priorities are assigned according to rate, with tasks having higher rates receiving higher priorities.

b. Straightforward scheduler: The basic rate monotonic algorithm requires only preemptive, static-priority-driven scheduling, which is commonly offered by operating systems. The rate monotonic algorithm is simply a method of assigning the static priorities for this form of scheduling so that all deadlines are met.

---

2. The earliest deadline and least slack time algorithms are optimal preemptive, *dynamic*-priority-driven scheduling algorithms [Liu and Layland 73; Mok 83].

c. Low overhead: The rate monotonic algorithm is low overhead in the sense that priority assignments are static; in other words, it is low overhead relative to dynamic-priority-driven algorithms.

d. Efficiency: The rate monotonic algorithm is *efficient* in two senses. First, it is optimal with respect to the class of static-priority-driven algorithms. Second, it is viewed as being competitive with respect to the class of dynamic-priority-driven algorithms. That is, the "overbuild" required to achieve a processor utilization of no more than $ln\ 2$ is generally regarded as being "acceptable."

e. Predictability: The rate monotonic algorithm is *predictable* in the sense that (under the specified conditions) deadlines are guaranteed to be met *a priori*. The need for exhaustive testing is eliminated.

f. Adaptability to modifications in the application: Because of its straightforward assignment of scheduling attributes and its predictability, the rate monotonic algorithm is tolerant of application modifications. That is, modifications do not cause undue hardship for application developers in terms of redesigning time lines or retesting, as they would in the commonly used cyclic executive approach [Baker and Shaw 89] to real-time scheduling. Application developers simply adjust the priority assignments to reflect any changes in the ordering of task arrival rates and confirm that the total utilization remains within specified bounds (e.g., less than $ln\ 2$).

g. Applicability to Ada real-time systems: Through their Real-Time Scheduling Theory in Ada Project, researchers at SEI have been working with Ada compiler vendors, real-time system developers, and government agencies to explore the feasibility of utilizing rate monotonic scheduling theory in the development of Ada real-time systems. Their conclusion thus far is as follows: ". . . it seems to be possible to support analytic scheduling algorithms [i.e., rate monotonic algorithms] in Ada by using an enlightened interpretation of Ada's scheduling rules together with a combination of runtime system modifications and appropriate coding guidelines" [Sha and Goodenough 90, 59]. The article from which this quote is taken details the scheduling rule interpretations, runtime system modifications, and coding guidelines that the SEI researchers have found to facilitate the use of rate monotonic scheduling theory in Ada real-time systems.

h. Extensibility: Perhaps the most important feature of the rate monotonic algorithm is its "extensibility." It has proven to be amenable to a number of extensions beyond traditional periodic task scheduling. For example, extensions

have been developed for dealing with aperiodic tasks [Sprunt 89], transient overload [Sha 87], and task synchronization [Rajkumar 88; Sha 88].

Together, these features make the rate monotonic algorithm an efficient and effective real-time scheduling algorithm for many applications.

3. IEEE REAL-TIME EXTENSIONS TO POSIX

A key component of a scientific foundation for real-time computing is a *real-time operating system*, i.e., an operating system designed to meet the unique demands of real-time computing. These demands include the following:

a. Mission-driven, application-directed resource management, rather than fairness-driven or efficiency-driven resource management.

b. Timely (i.e., both fast and predictable) response to both external and internal events.

c. Predictable (i.e, bounded) service times and overhead times.

d. Accurate time services that make time visible and accessible to applications.

Appendix A, "Real-Time Operating Systems: Overview of the State of the Practice," discusses these demands and how they are addressed by today's real-time operating systems.

An effort is now underway to define an open system standard interface for real-time operating systems. The effort is taking place under the POSIX (Portable Operating System Interface for Computer Environments) umbrella. POSIX is the product of IEEE Project P1003, which is sponsored by the Technical Committee on Operating Systems of the IEEE Computer Society. P1003 consists of a family of working groups, one of which is P1003.4, the Realtime Extensions Working Group. The POSIX working groups are defining interface standards based on UNIX®. All of the POSIX standards are intended to facilitate application portability at the source code level. While UNIX has become the operating system of choice on a large number of widely varying hardware bases, its proliferation of versions in fact impedes application portability. The POSIX working groups are chartered to remedy this situation by defining a *standard* operating system interface and environment based on UNIX.

The first POSIX working group, P1003.1, has produced a standard known as IEEE Std 1003.1-1990 (or POSIX.1 for short) in the IEEE standards community and as ISO/IEC 9945-1 : 1990 in the international standards community [ISO 90]. POSIX.1

® UNIX is a registered trademark of UNIX System Laboratories, Inc.

defines the interfaces to system services, including process management, signals, time services, file management, pipes, file I/O, and terminal device management. POSIX.1, in the UNIX tradition, is oriented toward the interactive time-sharing computing domain. Using POSIX.1 as a baseline, the P1003.4 Working Group is extending application portability to the challenging real-time computing domain.

Also in the UNIX tradition, POSIX.1 and the draft standards being developed by the P1003.4 Working Group are written in terms of C language bindings. However, there is ongoing work within IEEE Project P1003 to extend the POSIX standards to other languages, including Ada. The P1003.5 Working Group has produced a draft standard [IEEE 90a] that defines an Ada binding to POSIX.1. The draft standard is now in the ballot resolution process. Next, the P1003.5 Working Group plans to develop Ada bindings to the real-time extensions defined by the P1003.4 Working Group.[3]

The P1003.4 Working Group has achieved a broad base of participation and support. It includes over 175 individuals, representing over 70 organizations, including Intel, IBM, AT&T, DEC, General Motors, Hewlett-Packard, Motorola, U.S. Navy, NASA, Sun Microsystems, and Unisys. Moreover, the P1003.4 Working Group has done an effective job of capturing the best of the state of the practice in real-time operating systems. Once accepted as standards, the P1003.4 *Realtime Extension for Portable Operating Systems* [IEEE 89] and the P1003.4a *Threads Extension for Portable Operating Systems* [IEEE 90b] are expected to become both widely available and widely utilized.

Already, POSIX has been adopted by the National Institute of Standards and Technology (NIST) as a key component of its Applications Portability Profile (APP) [NIST 90], and by the Navy's Next Generation Computer Resources (NGCR) Program as a baseline on which to build its operating system interface standard [OSSWG 90]. Furthermore, POSIX has been selected for use in some specific large-scale systems, including the Space Station Freedom [Kovsky 90] and the Worldwide Military Command and Control System (WWMCCS) Automated Data Processing (ADP) System [DCA 89, Appendix R].

The POSIX standards—in particular, the two draft standards being formulated by Working Group P1003.4—hold promise for facilitating the development of strategic defense systems for two main reasons:

a. First, the two draft standards of Working Group P1003.4, when combined, form a reasonable competitor to other state-of-the-practice real-time

3. Professor Ted Baker of Florida State University has prepared an Ada binding to an earlier draft of P1003.4 [Baker 90]. Presumably, his binding will be used as a basis for the P1003.5 Working Group effort to produce a draft standard Ada binding to the POSIX real-time extensions.

operating system interfaces.

b. Second, many benefits can be accrued from the adoption of open system standards, including (1) connectivity and interoperability, (2) portability of programs, data, and people, including programmers, system administrators, network managers, operators, and users, (3) protection of software investment, which is a byproduct of portability, (4) and encouragement of commercial, off-the-shelf (COTS) acquisitions, with their attendant advantages in terms of "timeliness, cost, reliability, completeness of documentation, and training" [DSB 87, 3].

Synopsis of the P1003.4 and P1003.4a Draft Standards

To date, Working Group P1003.4 has prepared two draft standards. The P1003.4 draft standard defines application interfaces in ten functional areas, while the P1003.4a draft standard defines interfaces in an eleventh area—threads (i.e., lightweight processes). Following are brief descriptions of the interfaces in the eleven areas:

a. Timers: The P1003.4 draft standard defines interfaces to system-wide timers and to per-process interval timers. Through these interfaces, it makes time visible to processes, and it enable processes to schedule events in a variety of useful ways, including periodically. The data structure that is used by these interfaces to represent time provides for nanosecond resolution.

b. Priority Scheduling: The P1003.4 Working Group views preemptive, dynamic priority-driven scheduling as being fundamental in real-time systems. The P1003.4 draft standard defines two variants of preemptive, dynamic-priority-driven scheduling.[4] The variants are distinguished by the way in which processes of equal priority are scheduled. In the first variant, ready processes of equal priority are scheduled according to a first-in-first-out (FIFO) policy. In the second variant, ready processes of equal priority are scheduled according to a round-robin (RR) policy, with a specified time slice.

c. Shared Memory: The P1003.4 Working Group views the shared memory paradigm as being an important, traditional, high-performance mechanism for interprocess communication in real-time systems. The P1003.4 draft standard enables shared memory objects to be mapped into a process's virtual address space. Semaphores are envisioned as a mechanism for synchronizing access to shared memory.

4. It should be noted that preemptive, priority-driven scheduling, as defined in the P1003.4 draft standard, is fundamental to the application of rate monotonic scheduling theory.

d. Real-time Files: The P1003.4 Working Group views the capability of performing I/O operations with deterministic high performance as being crucial in real-time systems. It recognizes that contiguous files are a traditional mechanism for providing deterministic high-performance I/O, since most real-time systems utilize rotating magnetic disks as their file storage media. However, rather than providing a specific interface to contiguous files, it provides a more general interface to "real-time files." The approach taken to real-time files is to make some critical performance-related aspects of the operating system's implementation of files and I/O not only application visible but also to some extent application controllable. In particular, an application program can offer "hints" relating to characteristics of the application, which the operating system can take into account in making its resource management decisions.

e. Semaphores: The P1003.4 draft standard adopts the binary semaphore as the basic means of process synchronization. It notes that the binary semaphore is a "minimal" synchronization mechanism and that other mechanisms such as counting semaphores and monitors can be implemented on top of the binary semaphore.

f. Interprocess Communication Message Passing: The P1003.4 Working Group views the capability of passing messages with both deterministic and high performance as being crucial in real-time systems. The P1003.4 draft standard supports message passing as a form of interprocess communication. It provides for synchronous and asynchronous message transmission and receipt, prioritized messages, and multicast communication.

g. Asynchronous Event Notification: The P1003.4 Working Group recognizes the importance of asynchronous event notification. Moreover, it recognizes the following shortcomings of the POSIX.1 signal facilities as a mechanism for asynchronous event notification in real-time systems: (1) signals do not queue, so if two arrive before the first is handled, then the first will be lost, (2) signals cannot pass data, so they cannot readily indicate specific sources of events or errors, and (3) the number of possible user-defined signals is insufficient for many applications. The P1003.4 draft standard strives to overcome these deficiencies and to define a general-purpose, uniform, reliable interface that has both deterministic and high performance.

h. Process Memory Locking: The P1003.4 draft standard supports the notion that a process should be able to lock its address space, or specified regions

thereof, into memory. Such a capability is viewed as being crucial to deterministic high performance.

i. Asynchronous I/O: The P1003.4 draft standard provides a capability to perform I/O operations asynchronously. The motivation is to allow processes to perform multiple I/O operations concurrently with computations on I/O data.

j. Synchronized I/O: The P1003.4 Working Group recognizes that some applications, such as database applications, may require *assurance* of I/O completion. The P1003.4 draft standard refers to I/O that is to be done with assurance of completion as "synchronized I/O."

k. Threads: The P1003.4 Working Group recognizes that, for many applications, a fine-grained concurrency model provides a more robust paradigm for time-critical processing than the POSIX.1 process model. Such facilities, called lightweight processes or *threads* [Cooper and Draves 87; McJones and Swart 87], allow concurrency of portions of the application to be defined within a POSIX process, allowing such concurrent functions to share memory and other resources. The thread mechanism is particularly important in shared memory parallel processors and for Ada programs using the Ada tasking facility.

These interfaces are described in more detail in Appendix B, "Overview of the IEEE P1003.4 Realtime Extension to POSIX."[5].

In regard to the real-time computing demands identified at the beginning of this section, the interfaces defined in the P1003.4 and P1003.4a draft standards offer the following services and features:

a. Mission-driven, application-directed resource management: The P1003.4/.4a draft standards offer (1) preemptive, dynamic priority-driven scheduling, (2) process memory locking, (3) real-time files, (4) asynchronous I/O, and (5) synchronized I/O. Moreover, resolution of resource contention is consistently accomplished not at the discretion of the operating system, but at the *direction of the application*, through application-specified attributes such as process priority.

5. The article reproduced in Appendix B was published in early 1990. At the time that it was prepared for publication, the threads-related interfaces were a part of the P1003.4 draft standard. However, before the P1003.4 draft standard (Draft 9, December 1, 1989) was distributed for balloting, the threads-related interfaces were extracted; they were put into a new draft standard—P1003.4a—and then further developed during the course of the year. Draft 5 of P1003.4a (December 7, 1990) was distributed for balloting in January 1991.

For example, in addition to supporting preemptive, dynamic priority-driven scheduling for resolution of processor contention, the draft standard makes the following provisions: (1) processes blocked at semaphores are dequeued in priority order, (2) messages have a *type* field that can be used to establish priorities for message delivery, (3) events are grouped into event classes that can be used to establish priorities for asynchronous event notification delivery, and (4) asynchronous I/O operations can be assigned priorities.

Optional synchronization (i.e., process blocking) is offered consistently throughout the P1003.4/.4a draft standards. That is, a process can choose *not* to block when invoking an operation that might incur blocking. A conspicuous example is asynchronous I/O. Other examples include asynchronous message sending, asynchronous message receiving, conditional synchronous message receiving, and conditional invocation of semaphore operations.

b. Timely response to events: To support this, the P1003.4/.4a draft standards offer (1) high-performance interprocess communication in the form of shared memory and semaphores, as well as in the form of message passing, including a provision for optimized delivery of very short messages (i.e., a pointer's worth of information), (2) reliable, high-performance asynchronous event notification, and (3) threads (also known as lightweight processes).

c. Predictability of service times and overhead times: The P1003.4 Working Group philosophy here is to define metrics for each of its interfaces and to require vendors to report the values of the metrics. Thus, standard *metrics* are defined, but standard *values* are not. In addition, the asynchronous invocations and conditional invocations described above (in item a) enhance the predictability of an application's performance.

d. Time services: The P1003.4/.4a draft standards provide interfaces to fine-resolution timers that make time visible to processes and enable processes to schedule events in a variety of useful ways, including periodically.

Through these services and features, the P1003.4/.4a real-time extensions to POSIX can support aperiodic applications, as well as priority-driven periodic applications, such as those developed according to rate monotonic scheduling theory.

4. DISTRIBUTED REAL-TIME OPERATING SYSTEM R&D

In this section, the focus remains on real-time operating systems, but it shifts from the state of the practice, as represented by the POSIX P1003.4/.4a open system standards, to the state of the art, as represented by several ongoing R&D efforts on *distributed* real-time operating systems.

Distributed real-time operating systems strive to meet the objectives of both *distributed* operating systems and *real-time* operating systems. They are designed to support real-time computing on a platform comprising multiple computers interconnected by a high-performance network. Distributed real-time operating systems are becoming increasingly important because real-time computing systems are increasingly being implemented as networks of computers.

While the POSIX P1003.4 Working Group has been careful *not to preclude* distributed system realizations of the P1003.4/.4a interfaces, it has not yet attempted to provide full and direct support for such implementations. In the future, however, the POSIX P1003.4 Working Group may choose to enhance their draft standards with more direct support for distributed computing. The primary source of ideas for such support would be the ongoing R&D on distributed real-time operating systems.

In addition to support for distributed computing, most current real-time operating system R&D prototypes incorporate new resource management approaches, which deal with time and timing constraints explicitly (versus implicitly, for example, through priorities). Many current prototypes also incorporate new fault tolerance mechanisms. The concepts being explored in these prototypes hold promise for meeting some of the difficult problems encountered in the development of large, complex, real-time systems, which are epitomized by strategic defense systems.

4.1 BACKGROUND: DISTRIBUTED OPERATING SYSTEMS

Section 3 characterized *real-time* operating systems by identifying some of their unique services and features. This subsection characterizes *distributed* operating systems by distinguishing them from conventional centralized operating systems. It then goes on to discuss the potential of distributed operating systems for meeting some strategic defense system requirements.

4.1.1 Distinction between Distributed Operating Systems and Centralized Operating Systems

Both centralized operating systems and distributed operating systems are responsible for performing two basic functions: (1) managing "system resources" and (2) providing an effective "system interface" for users and applications. The difference between centralized operating systems and distributed operating systems lies in what constitutes the "system" on whose behalf the operating system is performing the functions. In the case of a distributed operating system, the "system" being served is really a "system of interconnected systems." The interconnection is typically achieved via local area network (LAN) technology. What the distributed operating system contributes is further "unification": it establishes one unified system out of many interconnected constituent systems.

To create a single-system illusion, a distributed operating system must, in effect, make the network that interconnects the constituent systems transparent; hence, distributed operating systems are said to offer "network transparency." The transparency manifests itself in a number of ways, for example, *global* user identification, *global* object naming, and *location-independent* interprocess communication.

A distributed operating system manages and controls constituent systems by running on each constituent system. Typically, a distributed operating system is structured as a kernel and a group of servers. The kernel, which implements some abstractions for processes and interprocess communication, resides and executes at each constituent system. The servers, which implement conventional operating system services such as a file service, can then run at only selected sites, according to the needs of the specific users or applications being served by the distributed system.

4.1.2 Role of Distributed Operating Systems in Strategic Defense Systems

Strategic defense systems and their development and maintenance systems will have complex architectures. They will exist on geographically and spatially distributed sites or platforms. A platform may contain one or more networks of processors (uniprocessors or multiprocessors).

Clearly, communication between processors is essential at many levels: between processors in a multiprocessor or parallel processor system, between systems on a LAN, between systems on wide area networks. Also, coordination of the processing activities is essential. Communication and coordination can be achieved in various ways. For example, in keeping with today's state of the practice, each system could have its own centralized operating system; communication could be achieved through standard

communication network protocols, and coordination (that might otherwise be provided by a distributed operating system) could be achieved by application programs.

Alternatively, today's state-of-the-art distributed operating systems could potentially be utilized to meet some communication and coordination requirements, especially at the level of communication and coordination among systems interconnected by a LAN. In particular, real-time distributed operating systems could be utilized in strategic defense systems, and general-purpose distributed operating systems could be utilized in SDIO development and maintenance systems.

By managing resources in a unified manner, a distributed operating system would offer some notable advantages: (1) the possibility of load distribution—shifting workload from heavily utilized processing nodes to lightly utilized nodes, (2) the possibility of parallel execution—executing (appropriately structured) applications on multiple processing nodes in parallel—thus aggregating the power of multiple small systems into a larger total system useful for some applications, and (3) the possibility of enhanced reliability and fault tolerance, through fault containment (by isolating faults within individual constituent systems) and fault recovery (by using constituent systems as redundant components). By further providing a unified system interface, a distributed operating system would relieve programmers and application programs from some of the communication and coordination burden—just as any operating system relieves both programmers and programs from performing many common chores.

4.2 SAMPLE OF ONGOING DISTRIBUTED REAL-TIME OPERATING SYSTEM R&D EFFORTS

At this point, it is too early to single out any specific distributed real-time operating system R&D effort as being the solution to the real-time problems faced by strategic defense systems. It is for this reason that the "promising development" cited in this paper is the *collective* work of the real-time operating system R&D community.

In September 1989, a Workshop on Operating Systems for Mission Critical Computing was held at the University of Maryland.[6] Although the organizers of the Workshop solicited papers on a number of topics, including reliability, fault tolerance, and security, the Workshop was dominated by real-time interests. Over two-thirds of the papers focused on real-time issues. Most current R&D efforts on real-time operating systems were in fact represented at the Workshop. Below, we offer brief summaries of several distributed real-time operating systems discussed at the Workshop. For further explanation

6. This Workshop was jointly sponsored by the Office of Naval Technology, the Office of Naval Research, the Space and Naval Warfare Systems Command, the University of Maryland Department of Computer Science and Institute for Advanced Computer Studies, and the Institute for Defense Analyses.

and details, the reader is referred to the proceedings of the Workshop [UOM 89].

Alpha (E. Douglas Jensen, Concurrent Computer Corporation)

Alpha is a distributed operating system designed to support mission-critical computing in large, complex, distributed systems. It is intended to embody mechanisms and policies to facilitate and enhance real-time responsiveness, reliability, fault tolerance, and security. With respect to real-time responsiveness, the Alpha approach is to manage resources according to application-specified timing constraints and semantic importance values on a system-wide best-effort basis. With respect to reliability and fault tolerance, the Alpha approach is to implement mechanisms for replication and atomic transaction features at the kernel level, and to implement policies at higher levels. With respect to security, the Alpha approach is to utilize capabilities for naming and protection. Alpha is object based. The conventional *process* abstraction is decomposed into two components: (1) a storage component—the *object* (essentially an instance of an abstract data type), and (2) a processing component—the *thread*. Threads move through objects via operation invocations.

ARTS (Hideyuki Tokuda, Carnegie Mellon University)

ARTS is a distributed real-time operating system implemented as part of Carnegie Mellon University's Advanced Real-time Technology (ART) project. It is designed to provide system developers and users an analyzable, predictable, reliable, real-time computing environment. ARTS is object based. Its objects can have worst-case execution times (i.e., *time fence* values) and timing-violation exception-handling routines assigned to their operations. Its threads can have scheduling attributes, such as semantic importance values and timing constraints, associated with them. When a real-time thread invokes an operation, a time fence protocol is used to determine whether the thread can meet its timing constraints on the operation. ARTS implements an Integrated Time Driven Scheduler (ITDS), which adheres to the principle of policy/mechanism separation, thus enabling a wide variety of scheduling policies to be supported. The ART project has produced two window-based tools: (1) *Scheduler 1-2-3*, a schedulability analyzer, and (2) *Advanced Real-time Monitor (ARM)*, a monitor/debugger that presents a visual image of runtime behavior. Additional topics being explored in the ART project's testbed include real-time communication and real-time data management.

CHAOS-ART (Karsten Schwan, Georgia Institute of Technology)

CHAOS-ART (Concurrent Hierarchical Adaptable Object System supporting Atomic Real-time Transactions) is a distributed real-time operating system designed to support adaptive hard real-time applications. The particular application that motivated its development is the Adaptive Suspension Vehicle, Ohio State University's six-legged walking machine. In the design of CHAOS-ART, attention has been focused on the interaction that occurs between the *subsystems* of a real-time system. The contention of the CHAOS-ART developers is that the interaction should reflect the interaction that occurs between the real-world entities represented by the subsystems. Since the form of interaction between real-world entities can vary, CHAOS-ART provides a variety of communication and synchronization mechanisms. CHAOS-ART is object based, so real-world entities are represented as *objects*, which interact via operation invocations. Invocations can be periodic or aperiodic, and they can have real-time attributes such as delays, deadlines, and criticalness associated with them. At a higher level, CHAOS-ART provides a mechanism referred to as an atomic real-time transaction. Such a transaction is a set of real-time invocations. For flexibility, CHAOS-ART transactions have three classes of attributes: real-time, concurrency control, and recovery. The values of the attributes can be varied to reflect a variety of high-level interaction semantics.

DRAGON SLAYER/MELODY (Horst Wedde, Wayne State University)

DRAGON SLAYER is a distributed real-time operating system, and MELODY is its adaptive file system. They are designed to operate in a hazardous environment; in particular, they are targeted to military land vehicles of the 1990s. Their goals include real-time responsiveness, reliability, fault tolerance, and graceful degradation. The DRAGON SLAYER/MELODY developers recognize that these goals can compete and conflict with one another. They have designed DRAGON SLAYER and MELODY to be able to adapt to changes in the environment and to achieve the desired balance among the competing goals. For example, files can be replicated for fault tolerance. But maintaining the consistency of the copies, through concurrency control algorithms, introduces overhead and delay, which can lead to missed deadlines. The DRAGON SLAYER/MELODY approach is to incorporate two levels of concurrency control, *strong* and *weak*. DRAGON SLAYER and MELODY monitor their environment and their own performance (in the form of an access history and missed deadline history). They then generate, relocate, or delete copies of files, and choose between strong and weak concurrency control algorithms, based on the results of the monitoring.

HARTOS (Kang Shin, University of Michigan)

HARTOS is an operating system designed for an experimental distributed real-time system called the Hexagonal Architecture for Real-Time Systems (HARTS). HARTS consists of multiprocessor nodes that communicate via a hexagonal mesh interconnection network, in which nodes have six neighbors. Each HARTS node has one or more application processors and a supporting network processor, which all run the HARTOS kernel and communicate via a backplane bus. Research is focused on fault-tolerant real-time communication and computing. The hexagonal mesh interconnection network, due to its inherent redundancy (in the form of multiple paths between pairs of nodes), provides a good foundation for fault tolerance. HARTOS is meant to build on this foundation. It provides location-transparent interprocess communication, with options for broadcast and multicast message delivery. It is intended to support process replication, through a *process group* mechanism, which in turn relies on the broadcast/multicast message delivery. Other research issues include fault-tolerant routing of messages through the hexagonal mesh, real-time scheduling of messages with deadlines, and clock synchronization.

Distributed iRMX (Timothy Saponas, Intel Corporation)

The Distributed iRMX operating system is the newest member of Intel's iRMX family of real-time operating systems. It runs on a distributed system consisting of multiple single-board computers (based on the Intel 386 microprocessor) interconnected via the MULTIBUS II message-passing backplane bus. Program development can be supported in this environment (i.e., in the same chassis) by a separate single-board computer running UNIX. Distributed iRMX is implemented as a layered operating system. The lowest layer is the kernel, iRMK, which performs task management, interrupt management, time management, and basic device management. The second layer is the nucleus, which provides a preemptive, priority-driven scheduler for real-time responsiveness, and a set of protection mechanisms (e.g., segmentation, privilege rings, isolated address spaces) for reliability and security. The nucleus also provides ports and mailboxes for intertask communication on an intraprocessor or interprocessor basis. The third layer is a network transport service, which provides OSI-based communication across a LAN. The fourth layer is a distributed I/O System, which provides a distributed real-time file system. Applications run as multi-tasking jobs, where jobs have isolated address spaces, and tasks run as lightweight processes within jobs.

Mach and Real-Time Mach (Richard Rashid and Hideyuki Tokuda, Carnegie Mellon University)

Mach is a distributed operating system kernel designed as a foundation for system software. The Mach kernel implements the basic abstractions of task (i.e., address space and port rights), thread (i.e., lightweight process, of which multiple can exist within a task), port (which play a naming and protection role similar to that performed by capabilities), message, and memory object. Different operating system environments can be built on top of these abstractions. Historically, the UNIX environment has been associated with Mach. Mach has provided a portable, high-performance platform for UNIX, enabling the wealth of UNIX software to run on a wide variety of advanced computer architectures. Due in part to its association with UNIX, Mach is aimed at interactive computing. Real-Time Mach is a version of the Mach kernel being developed for real-time computing, as part of CMU's ART project. Some of Mach's resource management policies are being modified to meet the real-time computing demand for predictability instead of the interactive computing demands for efficiency and fairness. Concepts from the ARTS operating system, including the ARTS real-time thread model and the ARTS scheduler (ITDS), are being incorporated into the Mach kernel. In addition, Mach is being integrated with the ARTS tools, *Scheduler 1-2-3* and *ARM*.

MARUTI (Ashok Agrawala and Satish Tripathi, University of Maryland)

MARUTI is a distributed real-time operating system designed for hard real-time, fault-tolerant, secure computing. MARUTI is object based. A MARUTI object has services that are invoked through service access points. An object also has a non-conventional component known as a joint. The joint holds static information giving resource, timing, fault tolerance, and security requirements for the services offered by the object. It also holds dynamic information, including a calendar that contains a temporal ordering of object services to be executed. MARUTI's approach to real-time computing is guaranteed-service scheduling. That is, once it accepts a job (defined by a computation graph giving services to be executed), MARUTI guarantees that the timing constraints of the job will be met with a specified degree of fault tolerance. MARUTI utilizes replication and consistency-control mechanisms to implement user-specified fault-tolerance constraints. It utilizes capabilities for protection and security. Currently, MARUTI is implemented on top of UNIX as a three-layered system (i.e., kernel, supervisor, and application), but a version in which the MARUTI kernel directly manages hardware resources is being developed.

Spring Kernel (John Stankovic and Krithi Ramamritham, University of Massachusetts)

The Spring Kernel is a distributed real-time operating system kernel designed for *next-generation* hard-real-time systems. It is implemented on a network of multiprocessors called SpringNet. Each node in SpringNet contains one or more application processors, one or more system processors, and an I/O subsystem. The system processors offload scheduling and other system functions from the application processors, and the I/O subsystem offloads some I/O, including intensive I/O from fast sensors. The application processors can thus be dedicated to real-time application tasks. In the Spring project, tasks are assumed to fall into three categories: critical, essential, and non-essential. Critical tasks are ones with hard deadlines; if a hard deadline is missed, catastrophic results might occur. In the Spring project approach, the hard deadlines of these tasks are guaranteed to be met by preallocating resources to their worst case. Essential tasks are ones that have deadlines and are important to the system, but do not lead to catastrophe if they miss their deadlines. Since there are many of these tasks, it is not feasible to preallocate resources to their worst case. The Spring project approach to these tasks is to establish on-line, dynamic guarantees for them if sufficient resources are available, or to provide early notification of imminent missed deadlines if resources are not available. Non-essential tasks then run in the background. Much effort in the Spring project has been devoted to the development of scheduling algorithms for this environment of mixed tasks.

StarLite Operating System (Robert Cook, University of Virginia)

The StarLite operating system is being developed in the context of the StarLite programming environment. This programming environment supports the development and execution of software for uniprocessors, multiprocessors, and distributed systems. Research topics being explored include prototyping and real-time operating system, database, and network technology. The StarLite operating system is designed to be adaptable and extensible. It is a layered system, with standard interfaces at each layer. A layer can have multiple implementations, each tailored to a certain environment, and each providing the standard interfaces. Layers are carefully defined, so that (1) individual layers (such as a file management layer) can be left out of a particular implementation if not needed and (2) individual layers can be implemented in hardware. A layered, real-time UNIX has already been implemented in the StarLite programming environment. A long-term goal of the StarLite project is to create an *operating system generator*, which would automatically select and compose implementations from a module library, based on input specifications describing the application requirements and the target architecture. The StarLite project is building a library of implementations suitable for real-time computing. Among recent additions to the library are real-time deadlock avoidance algorithms.

V (David Cheriton, Stanford University)

V is a distributed operating system designed as a general-purpose base for distributed computing, including real-time distributed computing. V is noted for its minimal kernel. The V kernel is said to act as a "software backplane": just as a hardware backplane provides slots, power, and communication, the V kernel provides address spaces, lightweight processes, and interprocess communication in the form of message transactions. Conventional operating system services, such as file management and printing, are provided by system servers, which are implemented above the V kernel at the user level as multiprocess programs. V is also noted for its interprocess communication, which offers fast response and several other features (e.g., datagrams, multicast delivery, prioritized message transmission and delivery, and conditional message delivery) that are viewed by the V developers as being keys to providing real-time responsiveness. Additional V mechanisms that facilitate real-time computing include strict priority-based scheduling, accurate time services, and the ability to lock programs in memory.

5. METHODS FOR ENHANCING REAL-TIME SYSTEM ROBUSTNESS BY TRADING PRECISION FOR TIMELINESS

Real-time systems are built to accommodate a specified level of workload. For some systems, this "built-to" level may represent an absolute worst-case workload. For most large complex systems such as strategic defense systems, however, the built-to level represents an *estimate* of the worst-case workload or some *acceptable fraction* of the known or estimated worst-case workload. In practice, the estimate or the acceptable fraction cannot be too high, or the system would be far too expensive; therefore, the built-to level actually falls short of the absolute worst-case workload for many real-time systems.

When the workload of an operational real-time system temporarily exceeds its built-to level, a "transient overload" is said to occur. A transient overload need not inevitably lead to total system failure. Indeed, a system that cannot tolerate any transient overloads would be too brittle to be useful in real missions. The more tolerant of transient overloads a real-time system is, the more *robust* and useful it is.

One approach to enhancing robustness with respect to transient overloads is to ensure that the deadlines or other timing constraints that are missed are those associated with the "least important" tasks. This approach is often taken in an *ad hoc* manner, simply by assigning priorities in accordance with importance. (Unfortunately, such priority assignments are suboptimal, and lead to unpredictable performance as well.) Rate monotonic scheduling theory takes a more disciplined approach through the period transformation method [Sha 87]. Similar approaches are taken in aperiodic real-time scheduling methods, such as the best-effort scheduling algorithm explored by Locke [Locke 86], in which workload is shed so that overall "value" to the system is maximized.

Another approach to enhancing robustness with respect to transient overloads is to trade precision for timeliness. In this approach, all deadlines are met, but they are met in a redefined sense. In particular, the computation time devoted to some tasks (prior to their deadlines) is reduced—at the cost of reduced precision in the results of the tasks. Thus, tasks meet their deadlines, but they do so by settling for approximate results rather than "exact" results.

Section 5.1 presents an example to help motivate the concept of trading precision for timeliness in the event of transient overload. In Sections 5.2 and 5.3, two methods of making the precision and timeliness tradeoffs are briefly described. The methods are discussed in detail in Appendix C, "Trading Precision for Timeliness: An Approach to the Development of Robust Real-Time Systems."

5.1 EXAMPLE

As an example, consider a fictitious autonomous reconnaissance drone. The drone has its own navigation and terrain avoidance system, and is designed to fly knap-of-the-earth to the target location, gather intelligence, and return to base. The system also has a rudimentary threat avoidance system that takes evasive action when it detects a missile lock. Under normal operating circumstances, the real-time system must perform the following periodic tasks: (1) maintain course by correlating midflight navigational corrections, (2) maintain cover while avoiding terrain, and (3) fine-tune the engine to conserve fuel. Since this is a periodic system, it is easily handled by a traditional real-time scheduler.

Unscheduled events, such as when an obstacle is detected ahead or when hostile missile lock is detected, tend to cause unpredictable levels of overload in the system, and overbuilding the system to handle all overload situations can be costly. An alternative approach is to allow the system to spend less time on operations of less importance by accepting less accuracy during the transient overload period. For example, if an unexpected obstacle is detected, precise fine-tuning of the engine can be sacrificed while more time is allocated to the terrain avoidance system. This action is still consistent with fulfilling the mission since the amount of fuel wasted in that short time is negligible, but crashing into obstacles will certainly end the mission. Similarly, when missile lock is detected, precise knap-of-the-earth terrain following can be sacrificed since maintaining cover is momentarily unnecessary.

5.2 BACKUP APPROXIMATION METHOD

The backup approximation method [Liestman and Campbell 80; Liestman and Campbell 86] seeks to ensure that all deadlines are met by augmenting primary tasks that cannot be reliably scheduled to meet deadlines with alternate tasks that *can* be reliably scheduled. The primary tasks provide exact solutions but have variable execution times, while the alternate tasks have predictable, bounded execution times but provide solutions of less accuracy. The alternate task is always scheduled, so that if the primary task misses a deadline, the scheduler can automatically substitute the approximate solution provided by the alternate task. Thus, all deadlines are met, with accuracy or precision traded for timeliness where necessary. Scheduling algorithms designed to support the

concept of backup approximations seek to ensure all deadlines are met while minimizing the number of alternate tasks that must be substituted.

5.3 IMPRECISE COMPUTATION METHOD

The imprecise computation method is the subject of a real-time computing research effort at the University of Illinois [Chung 90; Liu 89; Shih 89]. In this method, each task is broken up into two parts: a mandatory subtask that produces a rough (but in some sense acceptable) result, and an optional subtask that refines it. The deadlines for both mandatory and optional subtasks are the original task deadline. All mandatory subtasks are required to finish by their deadlines, with optional subtasks running in any left-over time before the deadline arrives. Additionally, all mandatory subtasks are required to complete before their optional counterparts can begin, since the optional subtasks use the results of the mandatory subtasks as the starting points of their computations.

The optional subtask must have the property that the accuracy of its result monotonically increases with the amount of time spent on the computation. This restriction allows the scheduler to increase the accuracy of any solution by allocating the optional subtask more time for computation. This restriction also ensures that the optional subtask can be preempted at any time and the best result so far computed will be available.

Algorithms Amenable to the Imprecise Computation Method

Trading precision for timeliness can be beneficial for many real-time applications where an imprecise but timely answer is more valuable than a late answer. The key to building such systems is to find ways to structure problems so that successive refinement algorithms, such as iterative bounding algorithms, can be utilized.

Iterative bounding algorithms converge to a solution by successive refinement of upper and lower bounds. The convergence may alternate between upper and lower bounds, as in a binary search algorithm, or it may proceed from a single bound.

Algorithms that converge by squeezing a solution between steadily approaching upper and lower bounds are referred to as *partitioning* algorithms. Examples of partitioning algorithms include sectioning algorithms such as Newton's method for finding the root of an expression, branch and bound techniques, well-behaved series expansions with alternating terms, binary and interpolation searches, and other searching techniques.

Accumulation algorithms are similar to iterative bounding algorithms in that they converge by successive accumulation of the solution. The difference is that one of the bounds is not well defined. The algorithm then works by accumulating from the defined bound. Many greedy algorithms fall into this category since they build solutions by

accumulation and do not reverse previous computations. Examples of this type of algorithms include minimal spanning tree algorithms, maximal graph matching by alternating path methods, minimal path algorithms, and series expansions with non-negative terms.

Iterative bounding algorithms are well suited for the imprecise computation method. The successive refinement nature of these algorithms makes them ideal candidates for use as optional tasks. The mandatory tasks can be computed either using the iterative algorithm with a fixed number of iterations, or using some other algorithm. Moreover, it is often possible to prove that the function of error in the solution over time has a particular shape (e.g., linearly decreasing, convexly decreasing). This information can be used in selecting appropriate scheduling algorithms [Chung 90].

6. PERFORMANCE ANALYSIS OF APERIODIC REAL-TIME SYSTEMS

In this section, an issue in need of increased R&D is raised. The issue is the performance analysis of aperiodic real-time systems, as well as of the aperiodic element of systems that are primarily periodic.

In general, development of real-time applications is undertaken in one of two divergent frameworks: a *periodic* framework or an *aperiodic* framework. In a periodic framework, an application is developed as a collection of periodic tasks, which are initiated at regular time intervals known as periods. In an aperiodic framework, an application is developed as a collection of aperiodic tasks, which are initiated at irregular time intervals in response to asynchronous events.

The periodic framework has the advantage that timing constraints are explicitly taken into account through the period mechanism. The end of each period is the hard deadline for the task initiated at the beginning of the period. Period lengths are chosen so that an application can "keep pace" with its environment. Since task arrivals are synchronized, a system can be sized to guarantee that all deadlines are met. Rate monotonic scheduling theory, discussed in Section 2 of this paper, affords an efficient, effective, disciplined approach to the development and sizing of periodic real-time systems.

The disadvantage of the *purely* periodic framework is that it does not accommodate asynchronous events (or aperiodic tasks), which inevitably occur in large complex real-time systems such as strategic defense systems. However, the periodic framework can be adapted to deal with asynchronous events in various ways. For example, a periodic task can be created to service asynchronous events (in which case the periodic task in effect *polls* for event occurrences), or asynchronous events can be processed as *background* tasks. Sprunt, Sha, and Lehoczky present methods for scheduling aperiodic tasks in a specific periodic framework—the rate monotonic framework [Sprunt 89]. Their methods provide lower average response times for aperiodic tasks than either polling or background processing, while at the same time maintaining guarantees of meeting all periodic task deadlines.

In an aperiodic framework, processing is event driven rather than periodic. Events occur asynchronously. Typically, priorities are used to establish a service ordering

for events. In general, the resource management objective is to process events as fast as possible, subject to their priorities.

In an aperiodic framework, it is difficult to accommodate periodic tasks. The problem is that the periodic tasks have *hard deadlines* and expect *100%* of them to be met. Unless sufficient processor time is *reserved* for periodic tasks, for example, by giving all periodic tasks higher priorities than all aperiodic tasks, then providing such deterministic assurance (of meeting 100% of the deadlines) is infeasible. Aperiodic tasks with hard deadlines suffer from the same lack of deterministic assurance, unless they are *well-behaved* in the sense of having some reasonable minimum separation time and can have sufficient processor time dedicated to them. As soon as a stochastic component (namely, aperiodic tasks or asynchronous events) is introduced into a system's workload, assurance of meeting deadlines must be cast in stochastic terms rather than absolute, deterministic terms. This point is illustrated in Appendix D, "Scheduling Aperiodic Tasks with Hard Deadlines in a Rate Monotonic Framework," which presents analytical results on cost and performance tradeoffs that must be made when aperiodic tasks with hard deadlines are introduced into a rate monotonic scheduling framework.

The shortcoming of the traditional (priority-driven) aperiodic framework is that it lacks mechanisms for dealing with time. Timing constraints are not specified, either explicitly or implicitly. Consequently, analytical methods are not available for ensuring that the timing constraints are met. Application developers are afforded little support in *sizing* the system to ensure that "as fast as possible" is indeed fast enough. They must rely on empirical performance evaluation methods, such as simulation and testing.

In efforts to address this concern, mechanisms for introducing timing constraints into aperiodic frameworks have been proposed and investigated, but analytical methods to support the development of aperiodic real-time systems utilizing these new mechanisms have not been developed. For example, time-value functions and best-effort scheduling can be utilized to achieve good results under stressful workloads [Jensen 85; Locke 86]; but before they can be effectively applied in the development of large complex real-time systems, specific guidance on how to assign time-value functions to tasks must be formulated, and analytical methods for estimating the performance actually yielded by best-effort scheduling must be developed.

Without analytical methods for evaluating performance, application developers will have to continue relying upon simulation and/or testing. Besides being costly, simulation and testing cannot provide the same kind of assurance as, for example, rate monotonic scheduling theory provides for periodic real-time system development.

The Office of Naval Research (ONR) recognizes this problem. Its purpose in launching its Accelerated Research Initiative on real-time computing was to establish a scientific foundation for developing real-time systems, both periodic-based and aperiodic-based. The ONR is addressing the problem, in part, through efforts in formal specification and verification [ONR 90].

While analytical methods for performance evaluation may not be able to provide the same kind of assurance as formal specification and verification can provide, they can potentially offer more assurance than simulation and testing at less cost than simulation, testing, or formal specification and verification. The model for these analytical methods is the body of queueing theory that has been developed for the analysis of general-purpose (i.e., non-real-time) computing systems. For general-purpose computing systems, queueing theory provides good results over a wide range of system and workload parameters for modest effort and resources. An analogous body of theory for aperiodic-based real-time systems would revolutionize real-time system development.

7. RECOMMENDATIONS

We conclude by making recommendations on each of the five topics highlighted in this paper:

Rate Monotonic Scheduling Theory

Recommendation: The SDIO should encourage the use of rate monotonic scheduling theory in the development of strategic defense real-time systems, in particular, those real-time systems that are periodic or primarily periodic. The SDIO should seek to do so through multiple means, such as education, training, policy, guidance, contractual requirements, and pilot projects.

Rationale: The rate monotonic scheduling algorithm and the theory that has evolved around it offer several important features, including (1) straightforward assignment of scheduling attributes, (2) straightforward scheduler, (3) low overhead, (4) efficiency, (5) predictability, (6) adaptability, (7) applicability to Ada real-time systems, and (8) extensibility. These features combine to yield an effective, efficient, *disciplined* approach to the development of periodic, as well as primarily periodic, real-time systems. A disciplined approach to development is essential for large, complex, mission-critical systems such as strategic defense systems.

IEEE Real-Time Extensions to POSIX

Recommendation: The SDIO should consider the adoption of open system standards, in particular, POSIX.1 and the real-time extensions to POSIX.1 that are embodied in draft standards P1003.4 and P1003.4a.

Rationale: Open system standards, in general, offer several benefits that could significantly enhance the producibility and affordability of strategic defense systems: (1) connectivity and interoperability, (2) portability of programs, data, and people, (3) protection of software investment, (4) and encouragement of COTS acquisitions. As widely supported open system (draft) standards, the IEEE real-time extensions to POSIX would enable SDI to reap these benefits. In addition to being open system standards, the real-time extensions to POSIX hold promise for

evolving into technically sound interfaces that effectively capture the best of the state of the practice in real-time operating systems.

Distributed Real-Time Operating System R&D

Recommendation: The SDIO should monitor the distributed real-time operating system R&D efforts taking place in government, industry, and academia, so that it can be prepared to exploit any breakthroughs that could facilitate the development, operation, and maintenance of strategic defense systems.

Rationale: The IEEE real-time extensions to POSIX are not, and do not pretend to be, the ultimate real-time operating system. Several ongoing real-time operating system R&D efforts are exploring new approaches to real-time computing. Many of the new approaches explicitly deal with time and timing constraints. In addition, the R&D efforts are addressing other issues of vital importance to the SDIO, most notably distributed system issues and reliability and fault tolerance issues. Therefore, it behooves the SDIO to keep abreast of developments in the real-time computing R&D community.

Methods of Trading Precision for Timeliness

Recommendation: The SDIO should exploit the concept of trading precision for timeliness to enhance system robustness. The SDIO should do so by (1) increasing awareness of the concept through education and training and (2) investigating, through funded R&D, the applicability of specific methods, such as the backup approximation and imprecise computation methods, to strategic defense system algorithms.

Rationale: In order for a strategic defense system or any other real-time system to be useful, it must be robust; specifically, it must be able to tolerate the transient overloads that will inevitably occur due to component failures, surges in threat levels, and variations in task computation times. Trading precision for timeliness can be an effective means of dealing with transient overloads. The backup approximation and imprecise computation methods represent disciplined approaches for making precision and timeliness tradeoffs.

Performance Analysis of Aperiodic Real-Time Systems

Recommendation: The SDIO should sponsor research on performance analysis of aperiodic real-time systems. The SDIO could solicit ideas and proposals through a mechanism such as the Broad Agency Announcement, which would encourage

innovation on the part of researchers and would allow the SDIO to selectively fund those proposals deemed to be the most promising in terms of meeting the specific requirements of strategic defense systems.

Rationale: Without analytical methods for evaluating the performance of aperiodic real-time systems, developers will have to continue relying upon costly, time-consuming methods such as simulation and testing.

REFERENCES

Baker 90 — Baker, T.P. 11 January 1990. *Realtime Extension for Portable Operating Systems Ada Binding*. Report for U.S. Army HQ CECOM, Center for Software Engineering.

Baker and Shaw 89 — Baker, T.P. and A. Shaw. 1989. The Cyclic Executive Model and Ada. *Real-Time Systems 1*, 1 (June), 7-25.

Brykczynski 90 — Brykczynski, B.R., C. Youngblut, and R.N. Meeson. 1990. *A Strategic Defense Initiative Organization Software Testing Initiative*. IDA Paper P-2493. Alexandria, VA: Institute for Defense Analyses.

Chung 90 — Chung, J. Y., J. W. S. Liu, and K. J. Lin. 1990. Scheduling Periodic Jobs that Allow Imprecise Results. *IEEE Transactions on Computers 39*, 9 (September), 1156-1174.

Cooper and Draves 87 — Cooper, E.C. and R.P. Draves. 2 March 1987. *C Threads*. Draft Paper. Pittsburgh, PA: Carnegie Mellon University, Department of Computer Science.

DCA 89 — Defense Communications Agency. November 1989. *WWMCCS ADP Modernization (WAM) Decision Coordinating Paper (DCP)*. Washington, D.C.: DCA.

DSB 87 — Defense Science Board. September 1987. *Report of the Defense Science Board Task Force on Military Software*. Washington, D.C.: U.S. Department of Defense.

Fife 87 — Fife, D. et al. October 1987. *Evaluation of Computer-aided System Design Tools for SDI Battle Management/C3 Architecture Development.* IDA Paper P-2062. Alexandria, VA: Institute for Defense Analyses.

Gordon and Linn 89 — Gordon, K.D. and C.J. Linn. April 1989. *Strategic Defense System Distributed Operating System R&D: Review and Recommendations.* IDA Paper P-2142. Alexandria, VA: Institute for Defense Analyses.

IEEE 89 — IEEE, Inc., Technical Committee on Operating Systems of the IEEE Computer Society. 1989. *Realtime Extension for Portable Operating Systems*, P1003.4/D9. New York, New York: IEEE, Inc.

IEEE 90a — IEEE, Inc., Technical Committee on Operating Systems of the IEEE Computer Society. 1990. *Draft Information Technology—Language Bindings to Portable Operating System Interfaces (POSIX)—Part 2: Ada*, P1003.5/D5. New York, New York: IEEE, Inc.

IEEE 90b — IEEE, Inc., Technical Committee on Operating Systems of the IEEE Computer Society. 1990. *Threads Extension for Portable Operating Systems*, P1003.4a/D5. New York, New York: IEEE, Inc.

ISO 90 — International Organization for Standardization (ISO)/International Electrotechnical Commission (IEC). 1990. *ISO/IEC 9945-1 : 1990 (IEEE Std 1003.1-1990), Information Technology—Portable Operating System Interface (POSIX)—Part 1: System Application Program Interface (API) [C Language].*

Jensen 85 — Jensen, E.D., C.D. Locke, and H. Tokuda. December 1985. A Time-Driven Scheduling Model for Real-Time Operating Systems. In *Proceedings of IEEE Real-Time Systems Symposium*, 112-122.

Kovsky 90 — Kovsky, Steven. 1990. NASA Chooses LynxOS For Space Station System. *Digital Review 7*, 2 (January 15), 49.

Liestman and Campbell 80 — Liestman, A. L. and R. H. Campbell. 1980. *A Fault-Tolerant Scheduling Problem*. Technical Report UIUCDCS-R-80-1010. Urbana, IL: University of Illinois, Dept. of Computer Science.

Liestman and Campbell 86 — Liestman, A. L. and R. H. Campbell. 1986. A Fault-Tolerant Scheduling Problem. *IEEE Transactions on Software Engineering SE-12*, 11 (November).

Liu 89 — Liu, J. W. S., K. J. Lin, C. L. Liu, and C. W. Gear. September 1989. Research on Imprecise Computations in Project QuartZ. In *Proceedings of the 1989 Workshop on Operating Systems for Mission Critical Computing*, University of Maryland, College Park, MD.

Liu and Layland 73 — Liu, C.L. and J.W. Layland. 1973. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *Journal of the ACM 20*, 1 (January), 46-61.

Locke 86 — Locke, C. Douglass. 1986. *Best-Effort Decision Making for Real-Time Scheduling*. Ph.D. Diss. Pittsburgh, PA: Carnegie Mellon University.

McJones and Swart 87 — McJones, P.R. and G.F. Swart. September 1987. *Evolving the UNIX System Interface to Support Multithreaded Programs*. Research Report 21, DEC Systems Research Center.

Mok 83 — Mok, A.K. 1983. *Fundamental Design Problems of Distributed Systems for the Hard Real-Time Environment*. Ph.D. Diss. Cambridge, MA: MIT, Department of Electrical Engineering and Computer Science.

NIST 90 — National Institute of Standards and Technology. 15 November 1990. *Application Portability Profile (APP): The U.S. Government's Open System Environment Profile*. Draft NIST Special Report. Gaithersburg, MD: National Institute of Standards and Technology, Systems and Software Technology Division, National Computer Systems Laboratory.

ONR 90 — Office of Naval Research. 25-26 October 1990. *Office of Naval Research Third Annual Workshop on Foundations of Real-Time Computing Research Initiative*, Washington, D.C.

OSSWG 90 — Operating Systems Standards Working Group (OSSWG). 1 June 1990. *Recommendation Report for the Next-Generation Computer Resources (NGCR) Operating Systems Interface Standard Baseline*. Compiled by D.P. Juttelstad. NUSC Technical Document 6902. Newport, RI: Naval Underwater Systems Center.

Rajkumar 88 — Rajkumar, R., L. Sha, and J.P. Lehoczky. December 1988. Real-Time Synchronization Protocols for Multiprocessors. In *Proceedings of IEEE Real-Time Systems Symposium*.

Sha 87 — Sha, L., J.P. Lehoczky, and R. Rajkumar. 1987. Task Scheduling in Distributed Real-Time Systems. In *Proceedings of IEEE Industrial Electronics Conference*.

Sha 88 — Sha, L., R. Rajkumar, and J.P. Lehoczky. 23 May 1988. *Priority Inheritance Protocols: An Approach to Real-Time Synchronization*. Pittsburgh, PA: Carnegie Mellon University, Departments of CS, ECE, and Statistics.

Sha and Goodenough 90 — Sha, L. and J.B. Goodenough. 1990. Real-Time Scheduling Theory and Ada. *IEEE Computer 23*, 4 (April), 53-62.

Shih 89 — Shih, W. K., J. W. S. Liu, J. Y. Chung, and D. W. Gillies. July 1989. Scheduling Tasks with Ready Times and Deadlines to Minimize Average Error. *ACM Operating Systems Review*.

Sprunt 89 — Sprunt, B., L. Sha, and J. Lehoczky. 1989. Aperiodic Task Scheduling for Hard-Real-Time Systems. *Real-Time Systems 1*, 1 (June), 27-60.

Stankovic 88 — Stankovic, J.A. 1988. Misconceptions About Real-Time Computing: A Serious Problem for Next-Generation Systems. *IEEE Computer 21*, 10 (October), 10-19.

Stankovic 89 — Stankovic, J.A., W.A. Halang, and M. Tokoro. 1989. Editorial. *Real-Time Systems 1*, 1 (June), 5-6.

UOM 89 — University of Maryland. 19-21 September 1989. *1989 Workshop on Operating Systems for Mission Critical Computing*, University of Maryland, College Park, Maryland.

Wheeler 90 — Wheeler, D.A. et al. 1990. *Reviews of Selected System and Software Tools for Strategic Defense Applications*. IDA-Paper P-2177. Alexandria, VA: Institute for Defense Analyses.

Youngblut 89 — Youngblut, C. et al. 1989. *SDS Software Testing and Evaluation: A Review of the State-of-the-Art in Software Testing and Evaluation with Recommended R&D Tasks*. IDA Paper P-2132. Alexandria, VA: Institute for Defense Analyses.

APPENDIX A

REAL-TIME OPERATING SYSTEMS:
OVERVIEW OF THE STATE OF THE PRACTICE

1. REAL-TIME COMPUTING SYSTEMS

Real-time computing systems are designated as such because of the significance of the role that the time dimension plays in them. In the premier issue of *Real-Time Systems*, the introductory editorial characterizes real-time systems as that category of systems in which "the correctness of the system depends not only on the logical results of computations but also on the time at which the results are produced" [Stankovic 89, 6].

Thus, in real-time computing systems, timeliness is mandatory. Timing constraints are imposed by the environment in which the real-time computing system exists. Typically, the environment consists of a larger *controlled* system (e.g., automobile, aircraft, ship, submarine, missile, hospital patient monitoring system, air traffic control system, factory floor, nuclear power plant, etc.), which is in turn embedded in and affected by its physical environment. The real-time computing system is the *controlling* system. Failure to meet environment-imposed timing constraints can have catastrophic consequences, such as loss of life, loss of the controlled system, or failure of the mission of the controlled system.

The qualifiers "hard" and "soft" are often used in conjunction with the term "real-time system." While precise definitions have not been agreed upon, the general distinction seems to lie in the nature of the timing constraints. *Hard real-time systems* have timing constraints that are both rigid and mandatory. For example, a task may have an absolute upper bound on response time that must never be exceeded. Such an upper bound is referred to as a hard deadline. *Soft real-time systems*, on the other hand, have more flexible timing constraints (sometimes referred to as soft deadlines). In the example, flexibility could entail (1) relaxing the requirement that the upper bound *never* be exceeded, by moving from deterministic performance specifications to stochastic specifications (e.g., response to an operator action must occur within 350 milliseconds with 97% probability), or (2) relaxing the upper bound itself, so that a response computed after the "upper bound" is still usable, although in some sense less valuable [Jensen 85; Locke 86], or (3) some combination of the above.

2. REAL-TIME APPLICATION DEVELOPMENT FRAMEWORKS

In general, development of real-time applications is undertaken in one of two divergent frameworks: a periodic framework or an aperiodic, asynchronous event-driven framework. In a periodic framework, an application is developed as a collection of periodic tasks. A periodic task is one that is initiated at regular time intervals, or *periods*.

The periodic framework has the advantage that timing constraints are explicitly taken into account through the period mechanism. The end of each period is the hard deadline for the task initiated at the beginning of the period. Period lengths are chosen so that an application can "keep pace" with its environment. Since task arrivals are synchronous, a system can be sized to guarantee that all deadlines are met. The disadvantage of the *purely* periodic framework is that it does not accommodate asynchronous events (or aperiodic tasks) which inevitably occur in all but the simplest control systems.

However, the periodic framework can be adapted to deal with asynchronous events in various ways. For example, a periodic task can be created to service asynchronous events (in which case the periodic task in effect "polls" for event occurrences), or asynchronous events can be processed as *background* tasks. In the June 1989 issue of *Real-Time Systems* [Sprunt 89], Sprunt, Sha, and Lehoczky present methods for scheduling aperiodic tasks in a specific periodic framework—the rate monotonic framework.[1] Their methods provide lower average response times for aperiodic tasks than either polling or background processing, while at the same time maintaining guarantees of meeting all periodic task deadlines.

In an aperiodic framework, processing is not primarily periodic but is instead event driven. Events occur asynchronously. Typically, priorities are used to establish a service ordering for events. In general, the resource management objective is to process events as fast as possible, subject to their priorities. In this type of framework, it is periodic tasks that are anomalous. The problem is not the fact that their interarrival times are constant but the fact that they have *hard deadlines* and expect *100%* to be met. Unless sufficient processor time is *reserved* for periodic tasks, for example, by giving all periodic tasks higher priorities than all aperiodic tasks, then providing such deterministic assurance is infeasible. Aperiodic tasks with hard deadlines suffer from the same lack of deterministic assurance unless they are *well-behaved* in the sense of having some reasonable minimum separation time and can have sufficient processor time dedicated to them. In general, as soon as a stochastic component (namely, aperiodic tasks or asynchronous

1. In a rate monotonic framework, tasks are assigned static priorities according to their arrival rates: tasks with higher rates are assigned higher priorities. Task scheduling is preemptive priority-driven. The rate monotonic algorithm was shown to be *optimal* among the class of preemptive, static priority driven algorithm by Liu and Layland [Liu and Layland 73].

events) is introduced into a system's workload, assurance must be cast in stochastic terms rather than absolute, deterministic terms.

The disadvantage of the traditional (priority-driven) aperiodic framework is that it lacks mechanisms for dealing with time. Timing constraints are not specified, either explicitly or implicitly. Consequently, methods are not available for ensuring that the timing constraints are met. The application developer is afforded little support in *sizing* the system to ensure that "as fast as possible" is indeed fast enough. In efforts to address this concern, methods for introducing timing constraints into aperiodic frameworks have been proposed and investigated [Jensen 85; Locke 86], but have not yet made the transition to become a part of the state of the practice in real-time computing.

3. OPERATING SYSTEM SUPPORT FOR REAL-TIME COMPUTING

As recently pointed out by Stankovic [Stankovic 88], today's real-time systems are built through brute-force techniques and at inordinate expense. As systems become ever larger and more complex, a more "scientific" approach is called for. An important ingredient of any scientific approach is an operating system designed to meet the unique needs of real-time computing, i.e., a *real-time operating system*.

Operating systems that claim to be real-time generally offer one or more of the following categories of services and features: mission-driven/application-directed resource management, timely response to events, predictable service times and overhead times, and time services that make time visible and accessible to applications. The significance of each of these categories is discussed below. It should be noted that most real-time operating systems are targeted to only one of the two real-time application development frameworks described above and that the importance of the different categories of services and features varies according to the target framework. For example, time services, specifically ones that enable periodic initiation of tasks, are more critical in a periodic development framework, while timely response to events is more critical in an aperiodic framework.

3.1 MISSION-DRIVEN/APPLICATION-DIRECTED RESOURCE MANAGEMENT

Over the past two decades, operating system research has been focused primarily on interactive computing. Common resource management goals have been to minimize average delay, maximize average throughput, and ensure "fairness" to competing users. While such efficiency-related and fairness-related goals may be well suited to the requirements of interactive computing, they do not adequately meet the requirements of

real-time computing. As discussed previously, real-time computing systems are distinguished by the presence of environment-imposed timing constraints.

A real-time operating system must be designed in accordance with the fact that a real-time computing system exists to perform a mission. The operating system should be supportive of the mission: the resource management provided by the operating system should be neither efficiency driven nor fairness driven, but *mission driven*. In particular, resource management should be driven by the time constraints of the mission, as conveyed to the operating system by the application. It is the responsibility of the application to specify resource management attributes to the operating system, and it is the responsibility of the operating system to manage *all* resources according to the application-specified attributes.

Mission-driven, application-directed resource management entails the following:

a. Application-directed allocation: In order to provide fast and predictable performance, a real-time operating system should enable an application programmer to specify certain "allocation attributes." For example, an application programmer might need to specify that a given program be memory resident or that a given file be contiguous.

b. Application-directed scheduling: Applications should have the capability to specify certain "scheduling attributes" that enable the operating system to impose an effective ordering on tasks or events. The operating system should take the scheduling attributes into account whenever contention or queueing occurs.

   The issue of exactly what the scheduling attributes should be is the topic of considerable controversy in the research community. Some believe that preemptive priorities are sufficient; others contend that the concept of priority must be broken down into complementary aspects of "urgency" and "importance," where urgency is meant to capture nearness of deadlines and "importance" is meant to capture criticality to the mission [Jensen 85]. A few go further, introducing even more resource management attributes, and making the scheduling problem even more complex.

c. Application-directed synchronization: The synchronization of concurrent activities should be controllable by an application. For example, an application should be able to specify whether operations (e.g., I/O, message passing) are to be done synchronously or asynchronously. While asynchronous operations introduce some complexity, they have been found to be useful in

real-time applications. The reasoning behind the provision of asynchronous operations is that synchronous operations may *unnecessarily* impede the forward progress of a path of execution.

3.2 TIMELY RESPONSE TO EVENTS

A real-time system must maintain its integrity with respect to the state of its environment. This can be viewed as a requirement to maintain *external* consistency, i.e., consistency between the actual state of the environment and the real-time system's perceived state of the environment. At the same time, *internal* consistency must also be maintained. That is, multiple concurrent tasks that constitute an application must have accurate perceptions of the states of one another.

If occurrences that alter the state of the environment or the system itself are viewed as *events*, then what is required is timely response to events. In other words, a real-time operating system should be able to respond to both external and internal events in a timely—both fast and predictable—manner; moreover, it should ensure that applications can also respond to events in a timely manner, through timely event notifications to applications.

3.3 PREDICTABLE SERVICE TIMES AND OVERHEAD TIMES

To facilitate the predictability of the performance of a real-time application, the execution times of operating system functions that the application explicitly invokes (via system calls), as well as those that it implicitly invokes, should be bounded. The bounds should not greatly exceed the means; otherwise, excessive resources may have to be dedicated to the application to assure acceptable performance.

3.4 TIME SERVICES

A real-time operating system should provide services that make time visible and accessible to applications. For example, applications should be able to set the time, read the time, and schedule events to occur at specified times, such as at periodic time intervals.

4. EXISTING REAL-TIME OPERATING SYSTEMS

4.1 TABLE-DRIVEN REAL-TIME EXECUTIVES: CYCLIC EXECUTIVES

Cyclic executives are designed to support a periodic application development framework. They are discussed in depth by Baker and Shaw in the June 1989 issue of *Real-Time Systems* [Baker and Shaw 89]. Briefly speaking, a cyclic executive has a single responsibility: to interleave the executions of periodic tasks according to a fixed,

predetermined schedule. The schedule is often specified in a static scheduling table, which indicates which pieces of which tasks are to be executed in what order during each time frame of each scheduling cycle. The scheduling table is formulated by the application developer as a key part of the application development. The formulation of a scheduling table can be viewed as a bin-packing problem, in which tasks are fit into cycles in such a way that all deadlines can be met.

The real-time services and features described above are addressed by cyclic executives as follows:

a. Mission-driven/application-directed resource management: Resource management decisions are deterministic. They are dictated by the application developer via the scheduling table.

b. Timely response to events: Asynchronous events and aperiodic tasks typically are handled by mechanisms such as polling or background processing. Timely response can be achieved through *frequent* polling.

c. Predictability of service times and overhead times: A cyclic executive is essentially a table-driven scheduler. Predictability of service times is a non-issue, since services (other than scheduling) are not offered. Predictability of overhead can be accomplished by straightforward measurement, since the only function of the cyclic executive—scheduling—is driven by static scheduling tables, which capture scheduling decisions made off-line by the application developer.

d. Time services: Periodic timer interrupts are vital to cyclic executives. They dictate frame boundaries and synchronize the system to the scheduling table.

4.2 PRIORITY-DRIVEN REAL-TIME EXECUTIVES

Stankovic and Ramamritham offer a concise description of today's priority-driven real-time executives in a paper presented at the 1987 Real-Time Systems Symposium [Stankovic and Ramamritham 87]. In the paper, they characterize most existing real-time executives or kernels as being "stripped down and optimized versions of timesharing operating systems." This is not surprising, given the previously noted fact that operating system research and development over the past two decades has been focused on the interactive or timesharing domain. In effect, familiar operating system concepts have been and are continuing to be used to construct new operating systems that can meet the unique demands of real-time computing. The two major design objectives are (1) minimization of overhead, which is the motivation behind the effort to "strip down" general-purpose timesharing operating systems, and (2) speed, which is the

motivation behind optimization efforts.

Stankovic and Ramamritham go on to enumerate several specific characteristics of today's priority-driven real-time executives [Stankovic and Ramamritham 87, p.146]. Below, we quote those characteristics in terms of our four general categories of real-time services and features:

a. Mission-driven/application-directed resource management: (1) priority scheduling, (2) the ability to lock code and data in memory,[2] and (3) the presence of special sequential files that can accumulate data at a fast rate.

b. Timely response to events: (1) the ability to respond to external interrupts quickly, (2) the minimization of intervals during which interrupts are disabled, (3) multi-tasking with task coordination being supported by features such as mailboxes, events, signals, and semaphores.

c. Predictability of service times and overhead times: (1) a small size (with its associated minimal functionality), (2) fixed or variable sized partitions for memory management (no virtual memory), and (3) a fast context switch.

d. Time services: (1) support of a real-time clock, (2) primitives to delay tasks for a fixed amount of time and to pause/resume tasks, and (3) special alarms and timeouts.

It should be noted that placement of the characteristics under the four service/feature categories involved some judgment calls, because some of the characteristics play roles in more than one category. For example, fast context switching contributes to timely response to events, as well as to predictability.

Priority-driven real-time executives as described above can support aperiodic real-time application development frameworks, as well as priority-driven periodic frameworks such as the rate monotonic framework.

4.3 PRIORITY-DRIVEN REAL-TIME OPERATING SYSTEMS

In this section, we discuss a class of real-time operating system that is closely related to the above class. The distinction between the two is that the real-time *executives* discussed above are intended solely for real-time computing, whereas the real-time *operating systems* of this section are, in a sense, general-purpose operating systems capable of meeting the demands of real-time computing.

---

2. This characteristic is quoted from a revised list that appears in the IEEE Computer Society Press's *Tutorial on Hard Real-Time Systems* [Stankovic and Ramamritham 88, p. 4].

Rather than speaking in generalities, we focus on the IEEE P1003.4 Realtime Extension for Portable Operating Systems (POSIX 1003.4). It is useful to do so, because POSIX 1003.4 arguably captures the state-of-the-practice in real-time operating systems.

Since POSIX 1003.4 is described in depth in Appendix B of this document, we simply put it into context here, by presenting its interfaces in terms of our four real-time service/feature categories:

a. Mission-driven/application-directed resource management: In this vein, POSIX 1003.4 offers (1) preemptive, dynamic priority-driven scheduling, (2) process memory locking, (3) real-time files, (4) asynchronous I/O, and (5) synchronized I/O. Additionally, POSIX 1003.4 attempts to take priorities into account in every interface in which contention or queueing can occur.

b. Timely response to events: To support this, POSIX 1003.4 offers (1) interprocess communication in the form of shared memory and semaphores, as well as in the form of message passing, (2) asynchronous event notification, and (3) threads, or lightweight processes.

c. Predictability of service times and overhead times: The POSIX 1003.4 philosophy here is to define metrics for each of its interfaces and to require vendors to report the values of the metrics. Thus, standard *metrics* are defined, but standard *values* are not.

d. Time services: POSIX 1003.4 provides interfaces to system-wide timers and per-process interval timers that make time visible to processes and enable processes to schedule timer events in a variety of useful ways, including periodically.

POSIX 1003.4, as well as priority-driven real-time operating systems in general, can support aperiodic real-time application development frameworks, as well as priority-driven periodic frameworks such as the rate monotonic framework.

REFERENCES

Baker and Shaw 89 — Baker, T.P. and A. Shaw. 1989. The Cyclic Executive Model and Ada. *Real-Time Systems 1*, 1 (June), 7-25.

Jensen 85 — Jensen, E.D., C.D. Locke, and H. Tokuda. December 1985. A Time-Driven Scheduling Model for Real-Time Operating Systems. In *Proceedings of IEEE Real-Time Systems Symposium*, 112-122.

Liu and Layland 73 — Liu, C.L. and J.W. Layland. 1973. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *Journal of the ACM 20*, 1 (January), 46-61.

Locke 86 — Locke, C. Douglass. 1986. *Best-Effort Decision Making for Real-Time Scheduling*. Ph.D. Diss. Pittsburgh, PA: Carnegie Mellon University.

Sprunt 89 — Sprunt, B., L. Sha, and J. Lehoczky. 1989. Aperiodic Task Scheduling for Hard-Real-Time Systems. *Real-Time Systems 1*, 1 (June), 27-60.

Stankovic 88 — Stankovic, J.A. 1988. Misconceptions About Real-Time Computing: A Serious Problem for Next-Generation Systems. *IEEE Computer 21*, 10 (October), 10-19.

Stankovic 89 — Stankovic, J.A., W.A. Halang, and M. Tokoro. 1989. Editorial. *Real-Time Systems 1*, 1 (June), 5-6.

Stankovic and Ramamritham 87 — Stankovic, J.A. and K. Ramamritham. December 1987. The Design of the Spring Kernel. In *Proceedings of IEEE Real-Time Systems Symposium*, 146-157.

Stankovic and Ramamritham 88 — Stankovic, J.A. and K. Ramamritham. 1988. *Hard Real-Time Systems*. Washington, D.C.: Computer Society Press of the IEEE.

APPENDIX B

OVERVIEW OF THE IEEE P1003.4 REALTIME EXTENSION TO POSIX[1]

1. INTRODUCTION

POSIX (Portable Operating System Interface) is the product of IEEE Project P1003, which is sponsored by the Technical Committee on Operating Systems of the IEEE Computer Society. Actually, P1003 consists of a family of working groups, one of which is the P1003.4 Realtime Extension Working Group. The POSIX working groups are defining interface standards based on UNIX.[2] All of the POSIX standards are intended to facilitate application portability at the source code level. While UNIX has become the operating system of choice on a large number of widely varying hardware bases, its proliferation of versions in fact impedes application portability. The POSIX working groups are chartered to remedy this situation by defining a *standard* operating system interface and environment based on UNIX. Notably, POSIX has been adopted as a key component of the Applications Portability Profile being developed by the National Institute of Standards & Technology (NIST).

The first POSIX working group, P1003.1, has produced a standard now known as IEEE Std 1003.1-1988, or POSIX.1 for short. POSIX.1 defines the interfaces to system services, including process management, signals, time services, file management, pipes, file I/O, and terminal device management. POSIX.1, in the UNIX tradition, is oriented toward the interactive time-sharing computing domain. Using POSIX.1 as a baseline, the P1003.4 Working Group is extending application portability to the challenging realtime computing domain.

The P1003.4 Working Group has achieved a broad base of participation and support. It includes over 175 individuals, representing over 70 organizations, including Intel, IBM, AT&T, Concurrent, DEC, General Motors, Hewlett-Packard, Motorola, NASA, Sun Microsystems, and Unisys. Once accepted as a standard, the P1003.4 realtime extension to POSIX is expected to become both widely available and widely utilized.

There are several other POSIX working groups, including P1003.0 (POSIX Guide), P1003.2 (Shell and Utilities), P1003.3 (Test Methods), P1003.5 (Ada Bindings), P1003.6 (Security), P1003.7 (System Administration), P1003.8 (Network Services), and

1. This paper appeared in *Real-Time Systems Newsletter 6*, 1 (Winter 1990), 9-18. The co-authors were William M. Corwin of Intel, C. Douglass Locke of IBM, and Karen D. Gordon of IDA.
2. UNIX is a registered trademark of AT&T.

P1003.9 (FORTRAN Bindings). In addition, there is a new standards committee, P1201.1, defining a standard windowing interface based on X Windows. The P1201.1 interface is being designed to work with any operating system; in particular, it is not dependent on POSIX, although it is anticipated that it will often be used in conjunction with POSIX.

The development of the P1003.4 draft standard realtime extension to POSIX has proceeded according to several important design guidelines. First, the P1003.4 draft standard, like POSIX.1, defines an application program interface to an underlying set of operating system functions; it does not specify the structure or functions of the underlying operating system beyond the specific functionality visible at the application program interface level. Second, the P1003.4 draft standard, again like POSIX in general, is intended to support application portability at the source code level and, in particular, *not* at the object code level. Third, the P1003.4 Working Group, in its charter, is constrained to make "the minimum syntactic and semantic changes or additions to the POSIX.1 standard in order to support portability of applications with realtime requirements." Fourth, in defining the P1003.4 interfaces, the P1003.4 Working Group has been careful to rely upon proven technology; consequently, the P1003.4 draft standard reflects the state of the practice in realtime operating systems. Fifth, recognizing the significance of performance in the realtime computing domain, the P1003.4 Working Group has defined standard performance metrics but not mandatory values for the metrics.[3] The idea is that vendors will be required to document the values of the standard metrics for their implementations. Then customers will be able to identify and acquire an implementation that meets their specific performance and cost requirements.

2. P1003.4 INTERFACES

In the view of the P1003.4 Working Group, the fundamental difference between capabilities required in realtime systems and those typically provided in time-sharing systems centers on the need of realtime systems for resource management that can support predictable (i.e., bounded) response times. In time-sharing systems, resource management objectives are efficiency and fairness. In realtime systems, on the other hand, resources must be managed so that time-critical application functions can control their response time, possibly resulting in delay or even starvation for non-time-critical application functions. Resources most frequently causing problems for realtime applications are processors, clocks, I/O interfaces and devices, communications, and memory. Therefore, the P1003.4 Working Group has focused its initial efforts on defining application

3. The reader is referred to the P1003.4 draft standard for listings and descriptions of specific metrics. Due to space limitations, we refrain from citing them here.

interfaces in the following eleven functional areas:

a. Timers

b. Priority Scheduling

c. Shared Memory

d. Realtime Files

e. Semaphores

f. Interprocess Communication Message Passing

g. Asynchronous Event Notification

h. Process Memory Locking

i. Asynchronous Input and Output

j. Synchronized Input and Output

k. Threads

A brief description of each of these interfaces is given below.

2.1 TIMERS

The P1003.4 draft standard defines interfaces to system-wide timers and to per-process interval timers that make time visible to processes and enable processes to schedule timer events in a variety of useful ways, including periodically. The data structure that is used by these interfaces to represent time provides for nanosecond resolution.

The P1003.4 timer facilities provide the following functions:

a. Setting the value of, getting the value of, and getting the resolution of a specified system-wide timer.

b. Creating and destroying a per-process interval timer, based upon a specified system-wide timer and a specified delivery mechanism (signals, events, or implementation specific). If events are specified, then the application programmer defines an event and writes an event trap routine in accordance with the P1003.4 asynchronous event notification facilities, which are described later in this section.

c. Setting the value of, getting the value of, and getting the resolution of a specified per-process interval timer. The "value" of an interval timer consists of

two parts: (1) timer interval and (2) remaining time to the next timer expiration. The remaining time to the next timer expiration can be set to a given offset from the current time (as known by the associated system-wide timer), or to a given absolute value. The timer interval, if nonzero, indicates that periodic timer expirations are to begin occurring after the initial timer expiration, where the period is equal to the specified timer interval.

2.2 PRIORITY SCHEDULING

The P1003.4 Working Group views preemptive, dynamic priority-driven scheduling as being fundamental in realtime systems. The P1003.4 draft standard defines two variants of preemptive, dynamic priority-driven scheduling. The variants are distinguished by the way in which processes of equal priority are scheduled. In the first variant, ready processes of equal priority are scheduled according to a first-in-first-out (FIFO) policy. In the second variant, ready processes of equal priority are scheduled according to a round-robin (RR) policy, with a specified time slice.

The P1003.4 scheduling facilities provide the following functions:

a. Setting the priority of and getting the priority of a specified process.

b. Setting the "scheduling policy" of and getting the scheduling policy of a specified process. The scheduling policy can be (1) preemptive, dynamic priority-driven, using FIFO within a priority level, (2) preemptive, dynamic priority-driven, using RR within a priority level, with a specified time slice, or (3) implementation specific.

2.3 SHARED MEMORY

The P1003.4 Working Group views the shared memory paradigm as being an important, traditional, high-performance mechanism for interprocess communication in realtime systems. The P1003.4 draft standard supports shared memory objects as "shared memory special files," i.e., objects named within the standard file system name space.[4] It enables shared memory special files to be mapped into a process's virtual address space. Semaphores are envisioned as a mechanism for synchronizing access to shared memory.

---

4. For reasons including uniformity and convention, all application-created objects that can be shared by two or more application processes are named within the standard POSIX.1 file system name space. In addition to shared memory special files, these objects include semaphore special files and message queue special files. In each of these cases, the use of the file system name space does *not* imply that the facilities (i.e., shared memory, semaphore, or message passing) need be implemented within the file management portion of the underlying operating system.

The P1003.4 shared memory facilities provide the following functions:

a. Creating a shared memory special file.

b. Opening and closing a specified shared memory special file.

c. Mapping and unmapping a specified segment of a specified shared memory special file into a process's virtual address space at a specified address.

The shared memory facilities are designed to be extensible, in that common global objects other than shared memory special files can be mapped into a process's address space through the shared memory facilities, in an implementation-specific way. The only restriction is that the objects be named within the file system name space. One specific extension singled out in the P1003.4 draft standard as being particularly relevant in the realtime computing domain is to enable a process to utilize the shared memory facilities to map sections of physical address space into its virtual address space.

2.4 REALTIME FILES

The P1003.4 Working Group views the capability of performing I/O operations with deterministic high performance as being crucial in realtime systems. It recognizes that contiguous files are a traditional mechanism for providing deterministic high-performance I/O, since most realtime systems utilize rotating magnetic disks as their file storage media. However, rather than providing a specific interface to contiguous files, it provides a more general interface to "realtime files." An implementation may choose to implement realtime files through contiguous files, but it is not forced to do so. An implementation is free to take advantage of advanced or non-traditional media that can provide deterministic high performance without relying on contiguity, in the framework provided by the P1003.4 realtime file facilities.

The approach that the P1003.4 draft standard takes to realtime file support is to make some critical performance-related aspects of the operating system's implementation of files and I/O not only application-visible but also to some extent application-controllable. In particular, an application program can offer "hints" relating to characteristics of the application, which the operating system can take into account in making its resource management decisions.

The P1003.4 realtime file facilities provide the following functions:

a. Creating a realtime file. In doing so, a process communicates to the system what it perceives as being *desirable* attributes of the realtime file, and it receives in return information on the *actual* attributes of the file as created by the system. As stated in the P1003.4 draft standard, it is the responsibility of

the application to compare all of the requested and returned attributes in order to ensure that a sufficient set have been honored.

b. Communicating to the system the desirable attributes of a specified (previously created) realtime file. Again, the specification of desirable attributes is just a *request* to the system. The process receives in return information on the actual attributes, and must decide how to proceed based on that information.

c. Getting the actual attributes of a specified realtime file or of realtime files of a specified file system.

d. Obtaining a suitably aligned buffer of a specified size either from a specified data area or from the system.

Performance-related implementation attributes include (1) amount of data to be transferred in realtime *I/O* operations, (2) space reserved for a file or file system, (3) whether or not file is to be extended, and (4) size of extents.

Hints, or advisory information, that an application can offer the operating system include the following: (1) file allocation should be optimized for sequential access, which can be taken to mean that contiguous allocation is desirable, (2) remaining file space should be zeroed upon truncation of the file, (3) reaccess of currently accessed data is unlikely, so a least recently used (LRU) caching policy might not be advantageous, (4) read-ahead might not be advantageous, and (5) application is doing its own caching, so system caching might not be advantageous.

2.5 SEMAPHORES

The P1003.4 draft standard adopts the binary semaphore as the basic means of process synchronization. It notes that the binary semaphore is a "minimal" synchronization mechanism and that other mechanisms such as counting semaphores and monitors can be implemented on top of the binary semaphore. It supports semaphores as "semaphore special files," i.e., objects named within the file system name space.

The P1003.4 semaphore facilities provide the following functions:

a. Creating a semaphore special file.

b. Opening and closing a specified semaphore special file.

c. Performing a *P* operation [Dijkstra 68] on a semaphore represented by a specified semaphore special file. The *P* operation can be invoked either unconditionally or conditionally. When invoked conditionally, the *P* operation is performed only if the semaphore is in an unlocked state, in which case the *P*

operation causes the semaphore to enter a locked state and the invoking process to become the holder of the semaphore. Blocked processes are granted the semaphore in priority order, with FIFO ordering for processes of equal priority.

d. Performing a *V* operation on a semaphore represented by a specified semaphore special file. The *V* operation can be invoked either unconditionally or conditionally. When invoked conditionally, the *V* operation is performed only if other processes are currently blocked by the semaphore.

Since semaphores have been defined as special files, it is possible for them to be opened by two processes that do not share physical address space if a distributed file system is utilized. The P1003.4 draft standard takes the following position on this point [P1003.4/Draft 8, p. 28]: "Semaphores are only required to operate when the processes using a common semaphore have the same physical address space. If a distributed file system is used, a mechanism shall be provided to ensure that only processes that have the same physical address space can access the semaphore."

2.6 INTERPROCESS COMMUNICATION MESSAGE PASSING

The P1003.4 draft standard supports message passing as a form of interprocess communication. The P1003.4 Working Group views the capability of passing messages with both high and deterministic performance as being crucial in realtime systems. The draft standard defines message passing in terms of "message queue special files," i.e., objects named within the file system name space. Message queue special files can be opened for use by multiple sending and receiving processes.

The P1003.4 message passing facilities provide the following functions:

a. Creating a message queue special file.

b. Opening and closing a specified message queue special file.

c. Sending a message to a specified message queue special file. The message to be sent is identified by a pointer (to a buffer in which the message is held) and a length. If the length is zero, then the pointer itself is the message, which facilitates an optimization for very short messages. This optimization entails passing the contents of the pointer to the receiver as the *event value* field in the asynchronous event notification associated with the message. This is an optimization in that the asynchronous event notification not only notifies the receiver of a message receipt, but also passes the message as part of the notification. It should be noted that the pointer's worth of information can indeed

be a pointer, for example, to a buffer in shared memory. The sender can specify that the message is to be sent either synchronously, in which case the sender blocks until the message is delivered to the receiver, or asynchronously, in which case the sender does not block. In the asynchronous case, the sender can further specify whether or not asynchronous event notification is to occur upon receipt. If the message is held in a buffer (as opposed to being a very short message entirely contained in a buffer pointer), then the sender can specify how the contents of the buffer are to be transferred—either by transferring control of the buffer to the receiver, by copying the contents of the buffer into an intermediate system buffer, or by granting access to the buffer to the receiver so that the receiver can copy the contents. Finally, the sender can associate a "type" with the message, which the receiver can use to selectively receive messages. The type can be used to prioritize messages, for example.

d. Sending a message to a specified list of message queue special files, thus providing a multicast capability.

e. Receiving a message from a specified message queue special file. The receiver can specify that the message is to be received asynchronously via an asynchronous event notification, or synchronously. If synchronous receipt is specified, then the receiver can further specify that the receipt is to be blocking (the receiver blocks until a message is available) or conditional (the receiver does not block, but only receives a message if one is available). The "type" field can be used to selectively receive messages in one of three ways: (1) FIFO delivery, (2) prioritized delivery, with FIFO for messages of equal priority (i.e., type), and (3) FIFO delivery of a specified type.

f. Allocating (by the sender) and freeing (by the receiver) a system-provided "message buffer" to hold the message, thus minimizing message copying.

g. Setting the values of and getting the values of attributes of a specified message queue special file. Attributes include the maximum number of messages, maximum number of bytes, whether or not to "wrap" new messages over old ones, etc.

h. Getting the status of a specified message queue special file.

2.7 ASYNCHRONOUS EVENT NOTIFICATION

The P1003.4 Working Group recognizes the importance of asynchronous event notification. Moreover, it recognizes the shortcomings of the POSIX.1 signal facilities as a mechanism for asynchronous event notification in realtime systems: (1) signals do not

queue, so if two arrive before the first is handled, then the first will be lost, (2) signals cannot pass data, so they cannot readily indicate specific sources of events or errors, and (3) the number of possible user-defined signals is insufficient for many applications. The P1003.4 draft standard strives to overcome these deficiencies and to define a general-purpose, uniform, reliable interface that has both deterministic and high performance.

The P1003.4 asynchronous event notification facilities consist of the following: (1) event definition data structure, (2) event trap routine function prototype definition, and (3) the functions cited below. In defining an event, a user specifies an event trap routine, an application-defined value to be passed to the event trap routine identifying the source of the event, the event class (i.e., a grouping of related events, including a priority in case of multiple event occurrences) within which the event trap routine executes, and the event class mask to be in effect during execution of the event trap routine.

Examples of asynchronous events include system-defined events specified in the P1003.4 draft standard, such as asynchronous I/O completion, timer expiration, and message arrival, as well as user-defined events. It should be noted that the P1003.4 asynchronous event notification facilities are purposefully not intended to support interprocess communication, although they are used as a delivery mechanism by the message-passing facilities.

The P1003.4 asynchronous event notification facilities provide the following functions:

a. Changing or examining the event class mask of the invoking process. Event classes that are included in the mask are blocked from being delivered via asynchronous event trap routines. Events are queued if they occur when masked.

b. Waiting for asynchronous event notifications for specified event classes, in one of two modes. In the first mode, the event processing is handled by the associated event trap routines; that is, the invoker is "enabling" the delivery of asynchronous event notifications to their respective trap routines. In the second mode, referred to as polling, the event processing is handled in-line; that is, the event notification (i.e., event class and application-defined event *value*) is delivered to the caller as part of the return from the invocation of the wait. The caller can specify that the wait be one of the following: (1) zero, in which case the caller does not wait, but only enables pending event notifications to be delivered, (2) indefinite, or (3) subject to a specified timeout period.

c. Causing a specified application-defined event to be raised for the invoking process.

d. Changing the number of queue entries to be used to hold events which have been raised but not yet delivered to the invoking process.

e. Achieving reliable exits from event trap routines via non-local jumps.

f. Associating a specified signal with a specified event class. This is a desirable, but not mandatory, capability that enables a deterministic delivery order to be achieved for signals.

Event classes establish a prioritization for event notification delivery. That is, if event notifications of different event classes are queued, then they are dequeued in order of event class, and in FIFO order for event notifications of equal event class.

2.8 PROCESS MEMORY LOCKING

The P1003.4 draft standard supports the notion that a process should be able to lock its address space, or specified regions thereof, into memory. Such a capability is viewed as being crucial to deterministic high performance.

The P1003.4 process memory locking facilities provide the following functions:

a. Locking and unlocking specified regions of a process's address space into memory.

It should be noted that the process memory locking interface defined by P1003.4 enables processes to specify certain "logical" regions of their address space for locking. These include the *data* region, the *text* region, the *stack* region, the *shared memory* region, and the *executable* region. However, the P1003.4 Working Group recognizes that some systems cannot support locking such memory regions separately; therefore, it makes the locking of logical regions optional.

2.9 ASYNCHRONOUS INPUT AND OUTPUT

The P1003.4 draft standard provides a capability to perform I/O operations asynchronously. The motivation is to allow processes to perform multiple I/O operations concurrently with computations on I/O data.

The P1003.4 asynchronous I/O facilities provide the following functions:

a. Asynchronously reading and writing a specified file. Control returns to the invoker when the read or write request has been initiated, or, at a minimum, when it has been queued to occur. The invoker can define an event and

associate it with the asynchronous I/O operation, in which case the event is raised upon completion of the operation. Alternatively, the invoker can poll for completion, by checking a field in the control block specified for the I/O operation. The invoker can also specify an asynchronous I/O operation priority to be used in determining the order of execution of asynchronous I/O operations with respect to one another. The significance of the priority is implementation specific.

b. Initiating a list of I/O requests with a single system call.

c. Cancelling a specified asynchronous I/O request, or cancelling all asynchronous I/O requests to a specified file.

2.10 SYNCHRONIZED INPUT AND OUTPUT

The P1003.4 Working Group recognizes that some applications, such as database applications, may require *assurance* of I/O completion. The P1003.4 draft standard refers to I/O that is to be done with assurance of completion as "synchronized I/O."

Two types of synchronization are defined in the P1003.4 draft standard:

a. Synchronized I/O *data* integrity completion: Completion is defined to occur for reads when data becomes available to the reading process. It is defined to occur for writes when both the data and the file system information necessary for retrieval of the data have been *successfully transferred.*

b. Synchronized I/O *file* integrity completion: Completion is defined to occur when, in addition to the above, *all* file system information relevant to the I/O operation has been *successfully transferred.*

Data is said to be *successfully transferred* "when the corresponding I/O peripheral in some implementation-defined fashion assures that all data is readable on any subsequent open of the file (even one that follows a system failure) in the absence of a failure of the physical storage medium" [P1003.4/Draft 8, p. 12].

The P1003.4 synchronized I/O facilities provide the following functions:

a. Specifying that I/O completion for a specified file is to be (re-)defined as either synchronized I/O data integrity completion or as synchronized I/O file integrity completion. A process can make this specification for a file that is being opened (as part of the POSIX.1 *open ()* function) or for one that is already open (via the POSIX.1 *fcntl ()* function).

b. Requesting that all outstanding I/O requests for a specified file are to be "completed" in accordance with the definition of either synchronized I/O data integrity completion or as synchronized I/O file integrity completion. The request can be either synchronous (i.e., control returns to the invoking process upon the completion of all the outstanding I/O operations) or asynchronous (i.e., control returns when the request is queued). In the asynchronous case, a specified asynchronous event notification will occur upon completion.

2.11 THREADS

The P1003.4 Working Group recognizes that for many applications, a fine-grained concurrency model provides a more robust paradigm for time-critical processing than the POSIX.1 process model. Such facilities, called variously light-weight processes or *threads* [Cooper and Draves 87] [McJones and Swart 87], allow concurrency of portions of the application to be defined within a POSIX process, allowing such concurrent functions to share memory and other resources. The thread mechanism has been identified as being particularly important in shared memory parallel processors and for Ada programs using the Ada tasking facility (refer to P1003.5 Ada Bindings Working Group).

The P1003.4 thread facilities provide the following functions:

a. Creating or *detaching* a thread, identifying a user-defined function to be executed concurrently with the current thread of control as a *thread*.

b. Synchronizing two or more threads sharing a common resource using a *mutex* (i.e., binary semaphore) or *condition* [Hoare 74].

c. Setting the scheduling policy and/or priority for a thread. The same scheduling paradigm defined under the P1003.4 Priority Scheduling functions is supported for threads. The semantics specified for thread scheduling provide that all threads compete equally based on their priorities; scheduling is not based on the enclosing process's priority.

d. Delaying a thread by a specified amount of time.

The P1003.4 draft standard also defines the mapping of events and signals to threads, as well as the effect of threads on POSIX system call error returns (i.e., *errno*). A consequence of using threads is that any previously defined system call that results in blocking the caller is modified to result in blocking only the calling thread.

3. CONCLUDING REMARKS

In summary, the interface defined by the P1003.4 draft standard offers several services and features that are considered essential in the realtime computing domain. First, it offers application-directed resource management in the form of preemptive, dynamic priority-driven scheduling, process memory locking, realtime files, asynchronous I/O, and synchronized I/O. Moreover, resolution of resource contention is consistently accomplished not at the discretion of the operating system, but at the *direction of the application*, through application-specified attributes such as process priority. For example, in addition to supporting preemptive, dynamic priority-driven scheduling for resolution of processor contention, the draft standard makes the following provisions: (1) processes blocked at semaphores are dequeued in priority order, (2) messages have a *type* field that can be used to establish priorities for message delivery, (3) events are grouped into event classes that can be used to establish priorities for asynchronous event notification delivery, (4) and asynchronous I/O operations can be assigned priorities. Also consistently throughout the P1003.4 draft standard, optional synchronization (i.e., process blocking) is offered. That is, a process can choose *not* to block when invoking an operation that might incur blocking. A conspicuous example is asynchronous I/O. Other examples include asynchronous message sending, asynchronous message receiving, conditional synchronous message receiving, and conditional invocation of semaphore operations.

Second, the P1003.4 draft standard strives to support timely response to internal and external events and errors. It offers (1) high-performance interprocess communication in the form of shared memory and semaphores, as well as in the form of message passing, (2) reliable, high-performance asynchronous event notification, and (3) threads, or lightweight processes.

Third, the P1003.4 draft standard addresses the issue of predictability of system service times and overhead times by defining standard metrics for each of its interfaces and requiring vendors to report the values of the metrics.

Fourth, as described in the previous section, the P1003.4 draft standard defines realtime-oriented timer facilities that provide interfaces to fine-resolution system-wide timers and per-process interval timers.

3.1 FORM OF THE P1003.4 STANDARD

The P1003.4 interfaces are system interfaces; as such, they constitute an extension to POSIX.1. However, the extension is not mandatory. The P1003.4 draft standard will be balloted as a set of *optional* interfaces.

To facilitate portability of realtime applications, the P1003.4 Working Group plans to define an Application Environment Profile (AEP) for realtime.[5] The Realtime AEP will single out some of the P1003.4 interfaces as being mandatory for the realtime computing domain. Strictly conforming realtime applications, namely, ones that utilize only mandatory interfaces, will be portable across operating systems that conform to the Realtime AEP. Participation in the P1003.4 Standardization Process

3.2 PARTICIPATION IN THE P1003.4 STANDARDIZATION PROCESS

There are three avenues of participation in the P1003.4 standardization process: (1) the Working Group, (2) the Balloting Group, and (3) independent reviews.

The Working Group is responsible for developing a draft standard. To accomplish this objective, it holds *open* meetings on a quarterly basis. The schedule for upcoming meetings is as follows:

January 8-12, 1990:	New Orleans, LA
April 23-27, 1990:	Salt Lake City, UT
July 16-20, 1990:	Danvers, MA
October 15-19, 1990:	Seattle, WA

The Balloting Group is responsible for reviewing and approving the draft standard prepared by the Working Group. Only when a specific percentage of Balloting Group members approves the draft does it become an IEEE standard. Participation in the Balloting Group is open to any member of the IEEE or the IEEE Computer Society. However, it is conventional practice that members of the Balloting Group also participate in Working Group meetings or independent reviews.

Independent reviews can be offered by any interested individual or organization by obtaining a copy of the current draft and providing comments in a form suitable for consideration by the Working Group.

3.3 STATUS OF THE P1003.4 STANDARD

The P1003.4 Working Group has been meeting quarterly since September 1987. Prior to that, many individuals now in the group had been meeting under the auspices of the /usr/group[6] Technical Committee Realtime Working Group.

As of December 1988, the Working Group had a complete draft of all functions intended for balloting. Since then, it has further refined and clarified the draft. The draft

5. The AEP concept is being considered by P1003 in general as a means of enabling operating systems to be tailored to specific application domains without sacrificing (intra-domain) application portability.

6. /usr/group is the International Network of UNIX System Users, an association of UNIX users.

has undergone extensive reviews by other groups, and the Working Group has resolved issues raised by those reviews.

The Working Group plans to begin the balloting process in January 1990 and to complete it by September 1990. It is anticipated that issues will be raised during the balloting process that require resolution before the draft can be declared an IEEE standard. The resolution process might entail a second round of balloting, which could last up to nine more months.

REFERENCES

[Cooper and Draves 87] Cooper, E.C. and R.P. Draves, "C Threads," Draft Paper, Department of Computer Science, Carnegie Mellon University, 2 March 1987.

[Dijkstra 68] Dijkstra, E.W., "Cooperating Sequential Processes," *Programming Languages* (F.Genuys, ed.), Academic Press, 1968, 43-112.

[Hoare 74] Hoare, C.A.R., "Monitors: An Operating System Structuring Concept," *Communications of the ACM 17*, 10 (October 1974), 549-557.

[McJones and Swart 87] McJones, P.R. and G.F. Swart, *Evolving the UNIX System Interface to Support Multithreaded Programs*, Research Report 21, DEC Systems Research Center, September 1987.

[P1003.4/Draft 8] P1003.4 Working Group, *Realtime Extension for Portable Operating Systems*, P1003.4/Draft 8, Technical Committee on Operating Systems, IEEE Computer Society, IEEE, August 1989.

APPENDIX C

TRADING PRECISION FOR TIMELINESS: AN APPROACH TO THE DEVELOPMENT OF ROBUST REAL-TIME SYSTEMS

1. INTRODUCTION

Real-time systems are systems whose correctness depends not only on the numerical integrity of a computation, but also on the time at which the computation is completed. Real-time systems have been traditionally difficult to design, and nearly impossible to verify because of the added complexity inherent in the timing interactions. In this paper, we address several new methods for scheduling real-time tasks by softening the requirements for numerical correctness. These methods have already met with some success, and may result in the ability to design and build systems that satisfy the timing constraints and have rigorously provable performance characteristics.

1.1 STRATEGIC DEFENSE SYSTEM REAL-TIME COMPUTING REQUIREMENTS

The real-time systems that constitute the Strategic Defense System (SDS) must satisfy at least three somewhat contradictory criteria. First, the systems must exhibit good behavior in overload situations. This means that the system must degrade gracefully as overload prevents certain deadlines from being met. A system that crashes helplessly on the first hit has little military value. Second, the system must perform predictably. Commanders must be able to make decisions with confidence that the system will be able to carry out the mission. Lastly, the system must be cost effective, and make the most use of the resources expended.

1.1.1 Overload Behavior

Good behavior under overload is important in battle situations where it is likely that the system will be heavily stressed. Penetration strategies may overload the system; equipment may be lost to random failure or attack; and communications may be delayed or denied. It is important that the system remain resilient under such stresses and maintain acceptable levels of performance.

1.1.2 Predictability

Aside from grace under pressure, system responses to fluctuations under stress must be well understood. That is, the expected performance under a wide range of stressful operating conditions must be predictable. This is important not only for commanders in an operational sense, but for political reasons as well. In the operational sense, commanders must be able to make informed choices about tactics and strategy when the system is stressed beyond its normal operating limits. In the political sense, funding decisions may well depend, in part, on the confidence that designers have that the system will be able to perform its mission in many situations. This kind of predictability requires a level of confidence in system performance in a variety of situations that is not presently possible with real-time designs.

Predictability is also important in the design phase as it allows engineers to make intelligent design choices. Knowing that a certain design has inherent predictability traits may allow designers to build more cost effective systems that more closely match operational requirements with a minimum of overbuild.

1.1.3 Cost Effectiveness

Cost effectiveness refers to attaining the highest levels of reliability and system performance with the least cost. Present approaches to reliability often rely on brute force redundancy, with the required performance often achieved by excessive overbuild. Given the projected cost and scale of the SDS, this reliance on massive hardware is neither practical or desirable. Aside from the obvious cost of redundant computing systems, additional penalties would be incurred due to the space based environment. These include additional power supplies, shielding, and associated lift costs. For the SDS, research should center on attaining high levels of reliability and performance without resorting to excessive redundancy, or overbuild.

1.2 MEETING THE REQUIREMENTS

These general SDS requirements can be described in terms of real-time systems. Good overload performance can be expressed as the ability to recover from missed deadlines by ensuring that the deadlines missed are the least important ones and by avoiding useless work. Such robust systems are able to manage missed deadlines to some degree without catastrophic consequences. Predictability translates into guarantees that can be made concerning the performance of the system. Lastly, cost effectiveness translates to the cost of the hardware required to do the job, and the percentage of time that the hardware sits idle. The goal is to minimize excess resources required to meet all other criteria.

Unfortunately, these three criteria have been difficult to attain in real-time systems. Good overload performance has traditionally come at the expense of lower processor utilization using the rate monotonic approach [Liu and Layland 73]. Algorithms that achieve optimal processor utilization, such as deadline driven algorithms, may not degrade gracefully under overload. Finally, performance under stress cannot always be guaranteed, and has been traditionally addressed by exhaustive testing with the hope that any flaws would be uncovered.

In the past, designers have worked around requirements incompatibilities by "softening" one or more operational criteria. In an ideal system, a designer strives for a system that simultaneously provides efficient processor utilization, while maintaining temporal and computational integrity. For SDS standards, the system should also be predictable in the sense that the performance is rigorously provable or verifiable.

While this list (i.e., high utilization, temporal integrity, computational integrity, and predictability) represents a difficult goal, softening any of the criteria does yiel ome design flexibility. For example, softening the utilization requirement by allowing less than 100% processor utilization yields the so-called rate monotonic algorithm. Liu and Layland [Liu and Layland 73] show that rate monotonic scheduling allows one to prove deadline satisfaction for a wide range of loads, with deadlines being missed under overload in a predictable fashion. Temporal integrity can be softened by recognizing that timing constraints can take forms other than hard pass/fail deadlines. This yields the so-called soft-deadline approach that has been successful in some situations where late computations can still be of some use.

It is also possible to soften the remaining two criteria. Computational integrity can be softened by trading computational accuracy for running time during periods of overload. This allows all deadlines to be met, although the system may operate somewhat less precisely due to inaccuracies. Predictability can be softened by allowing probabilistic performance guarantees to replace absolute performance guarantees. These probabilistic performance guarantees can be a result of stochastically defined workloads (i.e., interarrival times), or probabilistic guarantees on algorithmic performance such as those attainable for Monte Carlo or Las Vegas algorithms.

In the spirit of the soft-timing approach, these new approaches may be referred to as the soft-accuracy and soft-guarantee approaches, respectively. In this paper, we address the soft-accuracy approach for task systems where all arrival times and deadlines are assumed to be fixed. An issue involving the soft-guarantee approach under rate monotonic scheduling is addressed in Appendix D.

The remainder of this paper addresses two methods under the soft-accuracy approach that attempt to provide real-time scheduling systems with more tolerance to transient overload. The two methods are referred to as backup approximation and imprecise computation. Section 2 introduces the concepts of soft accuracy and outlines the two methods. Sections 3 and 4 present a detailed overview of the methods, and survey several scheduling algorithms for both periodic and general scheduling problems. Finally, Section 5 discusses applications that can benefit from soft-accuracy approaches. This section includes a discussion of general algorithmic techniques that can be used to build applications amenable to the soft-accuracy approach, and presents a specific application in distributed databases that uses the imprecise computation method.

2. SOFTENING COMPUTATIONAL INTEGRITY

In this section, we discuss the problem of scheduling real-time tasks on a set of one or more processors by softening computational integrity. The methods examined deal entirely with systems where all deadlines, ready times, and interarrival times are assumed to be fixed and have no stochastic variance. Methods are presented for dealing with both periodic systems, and general systems over a finite number of tasks with known ready times. No methods are presented for systems that exhibit stochastic (non-deterministic) interarrival times or ready times, as these systems would necessarily require both soft accuracy and soft guarantees.

In this section, we first introduce two models of scheduling real-time tasks, one for general tasks sets and one for periodic tasks sets. We then present an introduction to scheduling under soft-accuracy constraints and outline several different methods. The section concludes with an example emphasizing the soft-accuracy approach.

2.1 BACKGROUND

A real-time computing system is one in which the correctness of the result depends not only on the value returned, but on the time at which the result is produced. Such systems are typically used when a device must react to its environment. In general, a set of stimuli evokes system responses, and the system must react within a specified time period.

The physical stimulus-response problem described above can be abstracted into what is referred to as the general deadline scheduling problem. In this model, the concepts of stimulus, response, and timeliness are abstracted into sets of parameters that are more amenable to analysis. The general deadline scheduling model consists of a 4-tuple (T, τ, R, D), where $T = \{T_1, T_2, \ldots, T_n\}$ is a set of tasks with running times $\tau = \{\tau_1, \tau_2, \ldots, \tau_n\}$, ready times

$R = \{r_1, r_2, \ldots, r_n\}$, and deadlines $D = \{d_1, d_2, \ldots, d_n\}$. Precedence constraints $\prec$ may also be given to specify some partial ordering on the execution of the task set, where $T_i \prec T_j$ indicates that task T_i must complete before task T_j can be run.

The abstraction centers around modeling system responses as tasks. When a stimulus is detected the system reacts by scheduling a task to determine and execute the proper response. Each task $T_i \in T$ has a running time $\tau_i \in \tau$ which represents the amount of computing power required to complete the response. The stimulus concept is modeled by giving each task $T_i \in T$ a ready time $r_i \in R$. The ready time is the time at which the task becomes ready for processing, and no task may begin processing before its ready time. The timeliness of the response is enforced by assigning each task $T_i \in T$ a deadline $d_i \in D$. The deadline represents the time by which a ready task T_i must finish for the response to be timely.

The periodic scheduling problem is a specialization of the general scheduling problem in which all tasks are known to recur with a specific period. The system is formally represented as a 4-tuple (T, τ, P, D) where T is the set of tasks with running times τ. Each task recurs with period P and each occurrence must complete within D time units of the beginning of each period. It is often the case that $P = D$. This means that a task need only complete before the period expires and the next task arrives. As in the general scheduling problem, precedence constraints may also be specified which impose a partial ordering on task executions.

The goal of the real-time scheduler is to assign tasks to be processed so that some function $f(S)$ of the schedule S is maximized and the ready times and task precedence $\prec$ are not violated. The function f serves to measure the overall timeliness of the schedule and can take on many forms. In a hard deadline system f is a simple step function with value 1 when all deadlines are met and 0 otherwise. For soft deadline systems f may measure the total processing time spent past all deadlines, or other quantities.

2.2 SOFT-ACCURACY APPROACH

One idea that has recently been proposed to handle overload problems is to take advantage of the relationship between solution accuracy and computation time expended that is inherent in some algorithms. This concept has been explored using several methods. In this paper, we survey the backup approximation method and the imprecise computation method.

The backup approximation method [Liestman and Campbell 86] provides two tasks for every job in the system that may be prone to missed deadlines. The primary task is the full

implementation to perform the job, and the alternate task is a backup computation that yields an approximate result but in a predictable time. A real-time system is said to be in an erroneous state when the primary task for any job has not completed by its deadline. To recover from an erroneous state the system substitutes the output of the alternate task. The scheduler ensures that jobs can be satisfied with the alternate task if the primary task is unable to complete. Section 3 discusses the backup approximation method in detail and presents several algorithms for scheduling simply periodic systems.

The imprecise computation method [Liu 89] [Chung 90] provides two subtasks for each job. The first subtask, called the mandatory subtask, computes a rough estimate. The second subtask, referred to as the optional subtask, accepts a rough result and refines it to a more precise solution. The optional subtask must be such that the accuracy of the result monotonically increases with the amount of time spent on the computation. This is referred to as the monotone restriction and allows the scheduler to increase the accuracy of any solution by allowing the optional subtask more time. This restriction also ensures that the optional subtask can be preempted at any time and the best result so far computed will be available. Section 4 discusses the imprecise computation approach in detail and surveys several algorithms for scheduling both periodic and general scheduling problems.

Softening the accuracy requirement can be helpful in a number of ways, but the most studied effect is in overload management. In an overload situation, there is not enough processing power available to handle all of the outstanding requests, and hence some of the deadlines must slip. In a more traditional system some of the tasks would be entirely discarded because they had missed their deadlines, resulting in possibly catastrophic consequences. In the soft-accuracy approach, when an overload situation occurs the system merely softens the accuracy requirements on some of the tasks to buy more time. The old task set that exceeded processing capacity is effectively transformed into a new task set that is schedulable with the current processing capacity. As a result, all deadlines can again be met, and although some functions may be performed somewhat less accurately, no hard failures occur.

2.3 EXAMPLE

As an example, consider a fictitious autonomous reconnaissance drone. The drone has its own navigation and terrain avoidance system, and is designed to fly knap-of-the-earth to the target location, gather intelligence, and return to base. The system also has a rudimentary threat avoidance system that takes evasive action when it detects a missile lock. Under normal operating circumstances, the real-time system must perform the following periodic tasks: (1) maintain course by correlating midflight navigational corrections, (2) maintain

cover while avoiding terrain, and (3) fine-tune the engine to conserve fuel. Since this is a periodic system, it is easily handled by a traditional real-time scheduler.

Unscheduled events, such as when an obstacle is detected ahead or when hostile missile lock is detected, tend to cause unpredictable levels of overload in the system, and overbuilding the system to handle all overload situations can be costly. An alternative approach is to allow the system to spend less time on operations of lesser importance by accepting less accuracy during the transient overload period. For example, if an unexpected obstacle is detected, precise fine-tuning of the engine can be sacrificed while more time is allocated to the terrain avoidance system. This action is still consistent with fulfilling the mission since the amount of fuel wasted in that short time is negligible, but crashing into obstacles will certainly end the mission. Similarly, when missile lock is detected, precise knap-of-the-earth terrain following can be sacrificed since maintaining cover is momentarily unnecessary. Soft-accuracy systems seek to make such adjustments in a natural way with minimal overhead.

3. BACKUP APPROXIMATION METHOD

The backup approximation method described in [Liestman and Campbell 80] [Liestman and Campbell 86] is useful for scheduling independent task sets on a uniprocessor system. The method provides graceful degradation of service for simply periodic task sets[1] with stochastically distributed execution times without resorting to excessive overbuild. That is, the method is able to ensure that all deadlines are met for a wide range of processor loads while keeping processor utilization high in the average case.

The backup approximation method seeks to ensure that all deadlines are met by augmenting primary tasks that cannot be reliably scheduled to meet deadlines with alternate tasks that can be reliably scheduled. The primary tasks provide exact solutions but have variable execution times, while the alternate tasks have predictable, bounded execution times but provide solutions of less accuracy or precision. The alternate task is always scheduled such that if the primary task misses a deadline, the scheduler can automatically substitute the approximate solution provided by the alternate task. Thus all deadlines are met, with accuracy traded for timeliness where necessary. The algorithms in this section seek to ensure all deadlines are met while minimizing the number of alternate tasks that must be substituted.

The degree of scheduling reliability is a tradeoff between the uncertainty in the execution time of a task and the extent of overbuild that is acceptable in the system. The primary tasks cannot be reliably scheduled because their execution times vary widely. This is because

[1] Simply periodic tasks have periods that are integer multiples of the next smaller period.

a reliable schedule for the primary tasks would necessarily have to include enough idle time to handle all worst case running times. If this level of overbuild is not feasible, then the primary tasks cannot be reliably scheduled. The alternate tasks can be reliably scheduled with little average idle time since their runtimes are predictable, although a price is paid in the form of reduced solution accuracy. The backup approximation method thus guarantees deadlines without excessive overbuild by trading accuracy for more predictable runtimes when the need arises.

A backup approximation scheduling problem for a fault-tolerant (FT) system can now be formally defined as a 5-tuple (T, τ, A, a, P) where T is the set of primary tasks, and A are the alternate tasks. Each primary task T_i has runtime τ_i, and each alternate task A_i has runtime a_i. P is the set of periods which is the same for the primary and alternate tasks. The deadlines D are not included since they are assumed to be the same as the periods P. A task need only complete before the period ends and the next task arrives.

The optimality of the schedule is measured in terms of the number of alternate tasks that must be substituted as deadlines expire. If no alternate tasks are substituted in the schedule then all solutions delivered are exact and on-time, and the schedule is referred to as FT-optimal. If a schedule can be met by substituting any number of alternates then it is referred to as FT-feasible. The algorithms in this section produce FT-feasible schedules where the number of alternate results substituted is minimized.

3.1 SCHEDULING ALGORITHMS

This section introduces algorithms to build both optimal static and dynamic schedules for simply periodic task sets (defined later), where the optimality of the schedule is measured by the number of alternate (and hence imprecise) results that must be substituted in order to meet all deadlines. The static scheduling algorithm is optimal in that it substitutes the fewest alternate tasks for any precomputed schedule. The dynamic scheduling algorithm is optimal in that it substitutes the fewest alternate tasks for any possible schedule. The static algorithm will be presented first since the dynamic algorithm employs elements of the optimal static schedule.

Both algorithms require that the task system be simply periodic. A simply periodic task system can be described as a modification of the periodic scheduling problem as (T, τ, A, a, P, M) where T and A are the set of primary and alternate tasks with expected runtimes τ and precise runtimes a, periods P, and period multiples M. The simply periodic property is satisfied when all periods are integer multiples of the next smaller period as follows. Assume the task set is arranged with the periods in increasing order $\{p_1, p_2, \ldots, p_n\}$.

The set $M = \{m_1, m_2, \ldots, m_{n-1}\}$ is then the set of period multiples that defines period p_{i+1} from period p_i as $\forall i,\ p_{i+1} = p_i m_i$.

3.1.1 Static Algorithm

Let (T, τ, A, a, P, M) be a simply periodic FT system. A static schedule that maximizes the number of primary tasks completed can be calculated recursively using a greedy strategy. Assume that some optimal static schedule S_r has been calculated for tasks 1 through r where the tasks are numbered by increasing period. A provisional schedule S'_{r+1} is constructed by appending together m_r copies of schedule S_r. The schedule S'_{r+1} is then modified by removing copies of primary task T_p where τ_p is the largest among $\{\tau_1, \ldots, \tau_r\}$ appearing in schedule S_r. Copies of T_p are removed until enough idle time exists to schedule the alternate task A_{r+1}. If the resulting idle time is large enough, then task T_{r+1} is scheduled as well. Otherwise, if some primary task T_q exists in the schedule such that $\tau_q > \tau_{r+1}$, then replace T_q with T_{r+1}. The resulting schedule is S_{r+1}.

The above algorithm can be shown to maximize the number of primary tasks completed for any statically created schedule, and will have at least as much idle time as any such optimal schedule. Liestman and Campbell [Liestman and Campbell 80] prove that it runs in $O(\prod_{i=1}^{n-1} m_i)$ time for a schedule of n simply periodic tasks.

3.1.2 Dynamic Algorithm

In the static schedule shown above, it is often the case that an alternate task A_i and a primary task T_i are both scheduled. This is potentially wasteful since the alternate task will be run even if the primary task runs to completion. Thus the time spent on the alternate task was wasted and could have been used to run another primary task that was not able to complete in the static schedule. This is the basis for the dynamic backup approximation scheduling algorithm. Let S_n be a static schedule of n tasks. Create a revised schedule S'_n by scheduling all primary tasks before their alternates. When running the schedule, if any primary task T_i completes, the time reserved (later in the schedule) for its alternate A_i can be reallocated to other primary tasks. Thus the number of primary tasks that complete can be increased.

An algorithm using this strategy is presented in [Liestman and Campbell 86] and shown to maximize the number of primary tasks completed for *any* schedule with as much idle time as any such schedule. The algorithm is proven to run in $O(\prod_{i=1}^{n-1} m_i)$ time [Liestman and Campbell 86] to schedule n simply periodic tasks for one cycle of the largest period in P. A variation of the dynamic algorithm that uses a simple table is also possible (see

[Liestman and Campbell 86]). This variation is based upon a tree of schedules approach and provides faster reallocation online.

4. IMPRECISE COMPUTATION METHOD

The imprecise computation method is currently under development at the University of Illinois. In this method each task is broken up into two parts: a mandatory subtask that produces a rough, but acceptable result, and an optional subtask that refines it. The deadlines for both mandatory and optional subtasks are the original task deadline. All mandatory subtasks are required to finish by their deadlines, with optional subtasks running in any left over time before the deadline arrives. Additionally, all mandatory subtasks are required to complete before their optional counterparts can begin.

The system can now be formally defined as an extension of the general deadline scheduling problem. A general imprecise computation deadline scheduling problem is a 7-tuple (T, M, m, O, o, R, D). $T = \{T_1, T_2, \ldots, T_n\}$ is a set of tasks each of which is split into mandatory and optional subtasks $M = \{M_1, \ldots, M_n\}$ and $O = \{O_1, \ldots, O_n\}$ with runtimes $m = \{m_1, m_2, \ldots, m_n\}$ and $o = \{o_1, o_2, \ldots, o_n\}$. The mandatory and optional subtask sets have an intrinsic precedence relation $(\forall i)\ M_i \prec O_i$ that prevents any optional subtask from running before its mandatory counterpart has completed. The tasks become ready for processing at ready times $R = \{r_1, r_2, \ldots, r_n\}$, with deadlines $D = \{d_1, d_2, \ldots, d_n\}$.

As with traditional real-time systems there are scheduling algorithms to handle the general deadline scheduling problem and the periodic scheduling problem. The general schedulers presented in this paper are variations of a newly developed scheduling algorithm based upon network flow problems [Blazewicz and Finke 87], while the periodic schedulers presented are based upon traditional real-time schedulers using static and dynamic priorities. It should be noted that the algorithms presented in this section are sometimes suboptimal, even when the system is perfectly schedulable using traditional methods. However, these algorithms *are* useful when transient overloads occur frequently and overbuilding the system to the worst case is prohibitively expensive.

The error abating properties of the optional subtasks are important in the imprecise computation method. This is because scheduling is based upon the assumption of some sort of functional relationship between the error in the solution and the time spent computing. This assumption allows scheduling algorithms to measure the amount of error in terms of time spent in computation. In general, the imprecise computation method requires all optional subtasks to have the monotone error property. That is, the error decreases

monotonically as a function of computation time. Some scheduling algorithms in this section require even stricter error behavior such as linear dependence.

Even when the scheduling algorithm does not require the optional tasks to exhibit some specific error property, their error behavior may affect the total error of the scheduling algorithms. For example, an optional task may refine error more quickly in earlier iterations and more slowly in later iterations. Such tasks are particularly well suited to scheduling with the Least Attained Time algorithm, but do not do as well under the Earliest Deadline algorithm (both defined below).

The error properties referenced in this paper are assumed to have the form $\xi_i(\rho_j) = \left(1 - \frac{\sigma_{i,j}}{o_i}\right)^d$ where $\xi_i(\rho_j)$ is the error of optional task O_i in the j^{th} period, $\sigma_{i,j}$ is the processing time allowed task O_i in the j^{th} period, and d denotes the shape of the curve. Tasks that refine error more quickly in earlier iterations and more slowly in later iterations have the convex error property, denoted by $d > 1$. When $d = 1$ the error is reduced linearly, and is referred to as the linear error property.

4.1 SCHEDULING PERIODIC TASKS

The general imprecise computation scheduling problem can be recast as a periodic scheduling problem by simply replacing ready times with periods. The problem is a 7-tuple (T, M, m, O, o, P, D), where T is the set of tasks split into mandatory and optional subtasks M and O with runtimes m and o. Each task recurs with a period P and must complete within D time units of the start of the period. The optimality measure f is the average error over all tasks.

Periodic task sets are classified as type N or C based upon the system behavior under error accumulation. For systems of type N, tasks errors in consecutive periods can be tolerated as long as the average error over many periods remains low. For systems of type C tasks, the error is accumulated between periods, and a hard failure results if an optional task is not allowed to complete within some number of periods.

For example, frame processing of video images is a type N task while inertial guidance might be a type C task. In the video processing example, the frame stream is still useable even if each frame has a minor defect. This is because the error from previous frames has little or no effect on the error in subsequent frames. In the inertial guidance example, a positional error in one period is used as the starting location for the calculation in the next period and the errors in consecutive periods will tend to compound. Let us say the optional

subtasks must sometimes run to completion to eliminate the accumulated error, or a hard fault results.

Algorithms for scheduling both type N and C periodic systems make use of the rate monotonic algorithm of Liu and Layland. Traditionally this algorithm is used to trade some increased overbuild cost for deadline guarantees. It works by statically assigning priorities to tasks by decreasing interarrival rates, and scheduling tasks by these priorities at run time. Liu and Layland prove that the rate monotonic algorithm is able to guarantee all deadlines if the total processor utilization of the task set is less than approximately 69%. The total processor utilization is calculated as $U = \sum_{i=1}^{n} \frac{\tau_i}{p_i}$, where τ_i is the running time, and p_i is the period.

When applied to the imprecise computation method, the rate monotonic approach can be used to ensure that all mandatory subtasks meet their deadlines, with optional subtasks being scheduled by some other means. In all of the algorithms that follow, the priorities are partitioned into two classes. The mandatory tasks M receive rate monotonic priorities while the optional tasks O receive priorities based upon the particular algorithm being used. These optional task priorities may be static or dynamic. In either case the optional task priorities are set so that the highest priority of any optional task is lower than the lowest priority of any mandatory task. The mandatory and optional tasks are then scheduled according to these priorities. The priority queue is also purged of expired tasks since no value is added by executing beyond the deadline. The result is that all mandatory tasks take priority over optional tasks and are scheduled using the rate monotonic method and all optional tasks are scheduled in the idle time between mandatory tasks.

In the remainder of this section we will present several scheduling algorithms for both type N and type C tasks. When applicable, the results will be given for both uniprocessor and multiprocessor systems.

4.1.1 Scheduling Type N Tasks

Since type N systems are only affected by the average error a good performance measure would be the error of all tasks averaged over all periods. Given a feasible schedule S for task set T, let $f(S)$ be the error averaged over all tasks $T_i \in T$ for some time interval $[t_0, t_1]$. $f(S)$ can then be written as

$$f(S) = \frac{1}{|T|} \sum_{i=1}^{|T|} \frac{p_i}{t_1 - t_0} \left[\sum_{j=\ell}^{\ell + \lfloor \frac{t_1 - t_0}{p_i} \rfloor} \xi_i(\rho_j) \right]$$

where p_i is the period of task T_i, the index of the period starting at time t_0 is ℓ, and $\xi_i(\rho_j)$ is the error in subtask O_i incurred in the j^{th} period. Liu presents both static and dynamic priority driven schedulers that minimize or nearly minimize $f(S)$. Algorithms that minimize f are referred to as optimal, while all others are referred to as suboptimal.

4.1.1.1 Static Priority Algorithms

In this section we discuss algorithms for scheduling based upon the rate monotonic approach of Liu and Layland. These algorithms are called static because they rely on permanent priorities that are assigned to tasks when the system is built and do not change with time. Algorithms are presented for scheduling type N tasks on both uniprocessor and multiprocessor systems.

4.1.1.1.1 Least Utilization Algorithm

The least utilization algorithm is used to schedule periodic type N tasks on uniprocessor systems. The algorithm works by preemptively assigning the processor to the optional subtask that has the least utilization $u_i = \frac{o_i}{p_i}$. The algorithm can be proven optimal for a restricted subset of task sets when all optional subtasks have the linear error property (i.e., the error decreases linearly with computation time).

Priorities are computed as follows. Let the 6-tuple (T, M, m, O, o, P) represent a system of periodic tasks where T is a task set split into mandatory and optional subtasks M and O with runtimes m_i and o_i, and recurrence periods $p_i \in P$. Priorities are assigned based upon the value of $u_i = \frac{o_i}{p_i}$ for optional subtasks with the highest priorities going to subtasks with the lowest values of u_i. As with all imprecise periodic schedulers, all mandatory subtasks are assigned rate monotonic priorities with values higher than any optional subtasks. The mandatory and optional subtasks are then scheduled preemptively using these static priorities.

The linear error property is necessary but not sufficient to ensure optimal scheduling with the least utilization algorithm. That is, there are task sets that can be scheduled to achieve zero error that the algorithm will schedule with some non-zero error. Even when all periods are some integral multiple of the next smaller period (so-called simply periodic systems), the algorithm is suboptimal. Chung [Chung 90] shows an upper bound on the error for simply periodic task sets with weights w_i showing the relative importance of the tasks. The task set is required to have the linear error property and to satisfy $(\forall i \leq n)$ $s_i = p_i - \sum_{k=i}^{n} \frac{p_i}{p_k} m_k \geq o_i$. An upper bound on error for task sets scheduled using the least

utilization algorithm can then be written as

$$E < 1 - \sum_{k=1}^{i} w_k + \frac{w_i p_i}{p_{i-1}} \left\lceil \frac{\sum_{k=1}^{i}(m_i + o_i)}{p_i - \sum_{k=i}^{n} \frac{p_i}{p_k} m_k} \right\rceil$$

where i is the least index satisfying $s_i - \sum_{k=1}^{i-1}(m_i + o_i) \leq o_i$. If the task set is further restricted so that all tasks have the same period, then Chung is able to show that the least utilization algorithm minimizes the average error.

4.1.1.1.2 Extensions to Multiprocessor systems

The rate monotonic methods of Liu and Layland can also be used to schedule imprecise task sets on a multiprocessor system. In this approach all tasks are statically assigned to processors and each processor is scheduled using the rate monotonic algorithm. Assigning the tasks is similar to a bin packing problem and can be approximated using the First Fit Decreasing algorithm described below. The tasks are assigned to processors at initialization time, whereafter the processors schedule mandatory and optional subtasks from their individual task sets independently using the rate monotonic algorithm.

The First Fit Decreasing algorithm is used to assign tasks to processors. All tasks are ordered by increasing repetition periods and assigned to the lowest numbered processor (first fit) that can accept it. A task T_i can be accepted on a processor only if

$$\left(\sum_{T_k \in T'} \frac{m_k}{p_k} \right) + \frac{m_i}{p_i} \leq (n+1)(2^{\frac{1}{n+1}} - 1)$$

where T' is the set of tasks currently assigned to the processor and m_i and p_i are the mandatory runtime and period of the new task, and $n = |T|$. In [Chung 90], it is shown that this criterion is sufficient to guarantee all mandatory deadlines will be met.

The use of the rate monotonic algorithm to schedule tasks on individual processors implies some underutilization of processing capacity, and the resulting slack time is used to execute the optional tasks. In the imprecise computation method, this translates to guarantees that all mandatory deadlines will be met, and that there will be some guaranteed processor time left over for execution of optional tasks.

The general method described above can be modified to use any of the scheduling algorithms presented in the next section. The tasks are statically assigned to processors using the same First Fit Decreasing approach. The only difference is in how the priorities are assigned to the optional tasks when each processor begins scheduling its own set of assigned tasks.

4.1.1.2 Dynamic Algorithms

Dynamic scheduling algorithms rely on priorities that vary over time. Thus the system can adjust to changes in load and other unexpected events. Four such algorithms, the Least Attained Time, Best Incremental Return, Earliest Deadline, and Least Slack Time, are presented. Of these algorithms, only the Best Incremental Return algorithm is optimal, but is impractical since it requires perfect knowledge of the error behavior of all optional tasks.

4.1.1.2.1 Least Attained Time Algorithm

The Least Attained Time algorithm is used for scheduling on uniprocessor systems where optional tasks have the convex error property. That is, they refine error more quickly in earlier iterations and more slowly in later iterations. Processing time is assigned to optional tasks on a shortest-elapsed-time-first basis. That is, the task that has expended the least time has the highest priority.

The Least Attained Time algorithm is a preemptive, dynamic priority-driven algorithm that assigns priorities to optional subtasks based upon the amount of time they have been allowed to process. Optional tasks with the least accumulated processing time are assigned the highest priority, and optional subtasks that have logged the most processing time receive the lowest priority. As in the static algorithms, processing time is assigned to optional tasks only when there is no mandatory task available, and the processor may be preempted by any mandatory subtask that becomes eligible. This can be implemented as a preemptive, priority driven scheduler with dynamic priorities for the optional tasks and fixed, rate monotonic priorities for the mandatory tasks. The dynamic priorities are assigned to be lower than all fixed priorities to ensure that no optional task preempts a mandatory task.

No performance criteria have been proven for this strategy, but experiments in [Chung 90] seem to support intuition. For convex error functions, the Least Attained Time algorithm is among the best performers of the algorithms presented in this section, but performs more poorly for convex functions.

4.1.1.2.2 Best Incremental Return Algorithm

The Best Incremental Return algorithm is used for scheduling on uniprocessor systems, and can be shown to run optimally when all periods are equal. Unfortunately it requires perfect information about the error performance of the optional tasks, and for this reason it is considered impractical. As the name suggests it runs the optional task that promises the best incremental return during the next time quantum.

BIR is a greedy algorithm that schedules tasks by running the task that maximizes the amount of error reduction that can be attained in the next time slice. If the error function

$\xi_k(t)$ for optional subtask O_k is explicitly known, the expression for the incremental return can be calculated for each optional subtask that is ready. The incremental return for subtask O_k is then $\xi_k(\sigma_k + \delta) - \xi_k(\sigma_k)$, where σ_k is the accumulated processing time for task O_k and δ is the length of the time quantum.

This algorithm is impractical since it requires knowledge of the exact error behavior for all optional subtasks, and is included in Liu's study only as a benchmark. It is, however, possible to prove that this scheduling technique minimizes the average error when the periods of all tasks are the same.

4.1.1.2.3 Earliest Deadline Algorithm

The Earliest Deadline[2] algorithm schedules all optional subtasks based on the nearest deadline. Mandatory tasks are assigned fixed priorities using the rate monotonic formulation. To ensure that no optional task runs at the expense of a mandatory task, dynamic priorities are assigned lower values than all fixed priorities. Tasks are scheduled on a preemptive, dynamic priority driven approach.

In the imprecise computation method this algorithm can produce an optimal schedule if it exists when the total utilization U falls below 100%. Simulations in [Chung 90] show that the algorithm tends to perform better than other algorithms when the system is overloaded ($U > 1$), and the tasks have differing periods.

4.1.1.2.4 Least Slack Time Algorithm

The least slack time algorithm[3] attempts to estimate the urgency of the optional subtask by the amount of slack time left for that task. The slack time is defined to be the difference between the time to deadline and the projected processing time left in the task. The least slack time algorithm schedules all mandatory subtasks using the rate monotonic algorithm. When no mandatory subtask is outstanding, the algorithm preemptively assigns the processor to ready optional subtasks. The optional subtask with the lowest slack time receives the highest priority. As with the other algorithms this can be implemented as a dynamic priority driven scheduler where the mandatory tasks have fixed priorities that are higher than any assignable dynamic priority.

The least slack time algorithm is closely related to the earliest deadline algorithm. Simulations in [Chung 90] show that this algorithm performs similarly to the Earliest Deadline algorithm when the system is overloaded ($U > 1$), and the tasks have differing periods.

[2]In [Liu and Layland 73], the Earliest Deadline algorithm is proven to minimize tardiness in traditional deadline driven systems.

[3]For traditional deadline driven systems it is shown to optimally schedule any task set that is schedulable under the earliest deadline algorithm [Leung 89].

4.1.2 Scheduling Type C Tasks

Type C tasks accumulate error effects over consecutive periods, and fail either when the mandatory subtask fails to meet its deadline, or the accumulated error exceeds a specific threshold γ. The error accumulation is reversed only by allowing an optional task to run to completion so that a precise result is produced. For this reason schedulers for type C tasks must be able to guarantee that optional subtasks are able to run to completion on a regular basis, in addition to guaranteeing that all mandatory subtasks meet their deadlines.

In this section we consider scheduling of type C tasks where the error compounding is a fairly simple accumulation function. For an arbitrary task T_i, the error accumulation will be defined as the number of consecutive periods in which an error was produced since the last precise computation. The task fails when the accumulated error exceeds the threshold γ. This accumulation function has the result of requiring that for every γ periods at least one period produces a precise result, and any task T_k whose error accumulation exceeds its limit γ_k is defined to have failed.

4.1.2.1 Length Monotone Algorithm

In [Chung 90] it is shown the problem of scheduling optional tasks in an interval of γ periods into the future such that no task fails due to error accumulation is reducible to a multiprocessor scheduling problem and hence is NP-Complete. The length monotone algorithm seeks to heuristically build a feasible schedule for a task set T of type C algorithms on a uniprocessor system. That is, it seeks a schedule such that all mandatory tasks complete, and no task fails due to error accumulation. All tasks are assumed to have equal periods, and identical accumulation limits γ. A condition for guaranteeing schedulability of a task set due to Liu is also presented.

The length monotone algorithm works as follows. At the beginning of each period $\sum_{i=1}^{n} m_k$ units of time are allocated to cover all of the mandatory subtasks. If p is the period length, then $p - \sum_{i=1}^{n} m_k$ units of time are available in each period to allocate to optional subtasks. Call this the available period time. During any γ periods into the future there are $|T|$ remaining optional subtasks that must be completed to avoid a failure, one for each task. Let this set be R. Schedule the available period time for each of the γ periods into the future using a First Fit Decreasing algorithm (described below). If there is any time left over in any period, it is used to schedule additional optional subtasks to further reduce the error.

Tasks in R are scheduled for γ periods into the future using a First Fit Decreasing

algorithm [4] as follows. The optional tasks R are sorted in order of non-increasing execution times o_i. The sorted list is then assigned processing time in individual periods on a first fit basis. That is, assign the next task on the list to run in the first period in which it completely fits. In [Chung 90] it is shown that the algorithm is guaranteed to find a feasible schedule when

$$\frac{Q-1}{Q+1}u + \frac{2}{Q+1}U \leq 1$$

Here $U = \sum_{i=1}^{n} \frac{m_i+o_i}{p}$ is the total utilization factor, and $u = \sum_{i=1}^{n} \frac{m_i}{p}$ is the minimum utilization factor.

4.1.2.2 Extended Length Monotone Algorithm

The extended length monotone algorithm is a heuristic scheduling approach for uniprocessor systems where the tasks are simply periodic and have the error accumulation limits as described above. A task set is simply periodic when all periods are some integer multiple of the next smaller period in the task set. The algorithm assumes that the task set is partitioned into sets $T_1, T_2, \ldots, T_k$ where all tasks in a set T_i have the same period, ready time and deadline.

The algorithm works as follows. The subtasks in R are partitioned into sets $R_1, R_2, \ldots, R_k$ by identical periods. All mandatory tasks are scheduled according to the rate monotonic algorithm. The subtasks in R_1 are assigned to the processor using FFD in time intervals when no mandatory subtask is available. After the tasks in R_1 are scheduled the leftover time is used to schedule all tasks in R_2, and so on. No error bounds are known for this algorithm.

4.2 SCHEDULING GENERAL TASKS

The general deadline scheduling problem is a 7-tuple (T, M, m, O, o, R, D), where T is a set of tasks split into mandatory and optional subtasks M and O with runtimes m and o, ready times $r_i \in R$, and deadlines $d_i \in D$. An additional weighting function $w : T \mapsto \mathbf{R}$ can be specified to designate the relative importance of the tasks. The weighting function is used to emphasize the effect of individual tasks in the total error, and in this paper will be restricted to the special case of $\sum w_i = 1$. If no weighting function is specified, all tasks are assumed to have equal weight. Task sets are understood to have the precedence constraints between mandatory and optional subtasks, and there may also be additional precedence constraints $\prec$ specified on T. An additional 0/1 constraint may also be applied

[4] First analyzed by Johnson [Johnson 73] as a heuristic solution to the NP-Complete bin packing problem. He showed that the First Fit Decreasing algorithm guarantees solutions to within about 22% of optimal.

to the problem. The 0/1 constraint specifies that all optional subtasks must either run to completion, or their results are discarded.

The algorithms for scheduling general tasks are based upon the classical Earliest Deadline algorithm, and a more recently developed technique using network flow techniques [Blazewicz and Finke 87]. The network flow techniques are useful for uniprocessor scheduling of task sets with precedence constraints, and multiprocessor scheduling of independent task sets. The Earliest Deadline derivatives are somewhat faster for uniprocessor and multiprocessor scheduling, but require more restrictions on the task set.

All algorithms in this section require that all optional subtasks have a special linear error property where all subtasks have identical coefficients. That is, all optional subtasks produce results with error that decreases linearly with time spent on computation, and the rate at which the error is diminished is the same for all tasks. This allows the computational error ξ_i for optional task O_i to be approximated by $\xi_i = o_i - \sigma_i$, where σ_i is the time allocated to task O_i. Thus, the error is proportional to the time left to complete the preempted computation. The average error is then computed as $\sum \xi_i w_i$, where the w_i are the weights satisfying $\sum w_i = 1$. All algorithms in this section minimize the total error incurred over the schedule.

4.2.1 Network Flow Algorithms

The algorithms in this section are based upon finding the maximum flow in directed networks. In this technique an imprecise computation task system (T, M, m, O, o, R, D) is transformed into a flow network, and a schedule is determined by calculating the maximum flow through the network. These algorithms can be used to schedule dependent tasks on a uniprocessor system, or independent tasks on a multiprocessor system. For either case, systems with equal task weights can be scheduled in $O(n^2 \log^2 n)$ time, while systems with different task weights can be scheduled in $O(n^6)$ time. These network algorithms can also be used to schedule systems under the 0/1 constraint on a uniprocessor system when all optional subtasks have unit running time.

The networks are represented by directed graphs, or digraphs. A graph $G = (V, A)$ is a digraph with vertex set V and directed edge set A if all edges are directed. That is, if $(v_i, v_j) \in A$, then there exists a directed path from vertex v_i to v_j, but not necessarily in the other direction. All edges are labeled with a weight indicating the flow capacity of the edge in the direction specified. The digraph thus represents a system of flow capacities between vertices.

The flow problem is then to find the maximum legal flow capacity between any two vertices in the digraph. The vertex where the flow originates is referred to as the source, and the vertex where the flow terminates is referred to as the sink. A legal flow between vertices is one in which the flow on any edge does not exceed the specified capacity, and inflow equals outflow for all vertices except sources and sinks. At the source and sink, only the outflow and inflow capacities respectively are considered. Several fast algorithms exist to find legal flows in digraphs, most notable among these are the Ford-Fulkerson algorithms [Ford and Fulkerson 62] and related techniques.

4.2.1.1 Max-flow Algorithms

The max-flow scheduling algorithms can schedule unweighted task sets on uniprocessor and multiprocessor systems. Uniprocessor systems can be scheduled for dependent task sets with or without the 0/1 constraint. For task sets with the 0/1 constraint, all optional subtasks are required to run to completion in unit time, while no restrictions are placed on the task sets without the 0/1 constraint. For multiprocessor systems tasks can be scheduled only without the 0/1 constraint, but the task set must be independent. In all cases the algorithm runs in time $O(n^2 \log n)$.

The digraphs constructed for these scheduling algorithms all follow the same general layout. They all contain a single source and sink, with no edges flowing into the source, or out of the sink. Vertices are created to represent all tasks T_i, subtasks M_i and O_i, and time intervals A_j available for scheduling. A few additional vertices are provided in the construction to represent the source S_1, sink S_2, and a special vertex, I, used to measure schedulability. Directed edges are laid between vertices using running times o_i and m_i, and time intervals as weights. A special edge λ connects the vertex I to the sink S_2, and is varied during the algorithm to measure schedulability.

To schedule dependent task sets on uniprocessor systems, the algorithms make use of modified deadlines defined as follows. Let $T_1, \ldots, T_n$ be a set of tasks with deadlines $d_1, \ldots, d_n$. T_j is defined to be a successor of T_i if $T_i \prec T_j$ in the precedence ordering. Let A_i be the set of all successors for task T_i. The modified deadline d_i' is then defined as $\min_{T_j \in A_i}(d_i \cup d_j)$. Lawler and Moore [Lawler and Moore 69] proved that a feasible uniprocessor schedule exists using the modified deadlines D' if and only if a schedule exists using the unmodified deadlines D.

This result allows the precedence constraint to be ignored by computing schedules with the modified deadlines. When such a schedule is found, it can be rearranged into a legal schedule under the precedence by exchanging the assigned times for offending tasks until

the precedence constraints are satisfied. The resulting schedule then satisfies the precedence while preserving the same overall error.

An example digraph is shown in Figure 1.

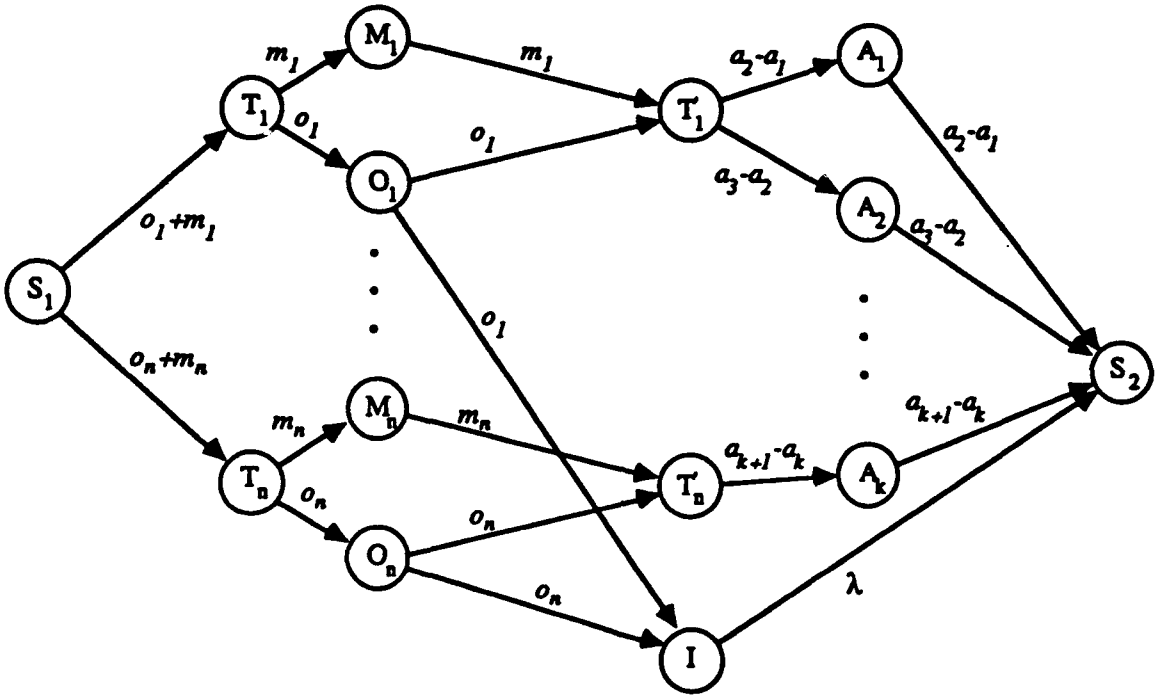


Figure 1. Digraph Representation of Task System (T, M, m, O, o, R, D)

It consists of vertices

$$\{T_1, \ldots, T_n, T'_1, \ldots, T'_n, M_1, \ldots, M_n, O_1, \ldots, O_n, S_1, S_2, I, A_1, \ldots, A_k\}$$

where the T, T', M, and O vertices correspond to tasks T, M, and O respectively, S_1 and S_2 are the added source and sink, and I is the added special vertex. The A vertices correspond to time intervals, and are constructed as follows. Let $\{a_1, a_2, \ldots a_{k+1}\}$ be the combined set of deadlines and ready times $D \cup R$ arranged in increasing order. Now each pair (a_i, a_{i+1}) represents a time interval, and the sequence of intervals represented by $\{a_1, a_2, \ldots a_{k+1}\}$ covers the entire scheduling interval. Vertex A_i corresponds to the time interval (a_i, a_{i+1}).

Edges labeled with flow capacities are laid between the vertices as follows. For all T_i, an edge is laid between S_1 and T_i, and carries weight $m_i + o_i$. Each T_i is separately connected to vertices M_i and O_i with weights m_i and o_i respectively, while each M_i and O_i is in turn connected to T_i', again with weights m_i and o_i respectively. In addition, all O_i vertices connect to I with weights o_i. Each T_i' vertex is connected to each A_j vertex for which task T_i is schedulable in the interval $[a_j, a_{j+1}]$ with weight $a_{j+1} - a_j$. Each A_i vertex connects to S_2 with weight $a_{i+1} - a_i$, and I connects to S_2 with weight λ. A more complete description of the digraph construction and associated algorithms can be found in [Shih 89a].

The basic algorithm is used to find uniprocessor schedules with minimal total error for unweighted task sets without the 0/1 constraint. It works by constructing the digraph representation of the problem and measuring the maximum flow attainable from S_1 to S_2 for various capacities of edge λ. The flow is first measured with $\lambda = 0$. Since the flow out of vertex I over edge λ is 0, the total flow into I is also 0. So this flow is the maximum amount of time that can be allotted to all tasks in an optimal schedule, and can be used to determine the amount of total error $\xi = \sum_{i=1}^{n} \xi_i$ in an optimal schedule. The flow is then measured with $\lambda = \xi$, and a second algorithm uses the result to create a preemptive schedule with total error ξ. The complexity of the basic algorithm is $O(n^3 \log n)$, and can be improved by minor modifications to achieve time $O(n^2 \log n)$.

The basic algorithm can be extended to schedule unweighted task sets on a uniprocessor with the 0/1 constraint if all optional tasks have unit running time. The schedule then produces minimal error by minimizing the number of incompleted optional subtasks since they do not reduce error. The extended algorithm first uses the basic algorithm to find a schedule with minimal error ignoring the 0/1 constraint. This schedule is then adjusted by trading time slots for tasks until the number of incompleted optional subtasks is optimized. The extended algorithm also runs in time $O(n^2 \log n)$.

Multiprocessor scheduling requires that the unweighted task set be completely independent except for the implicit precedence relation between the mandatory and optional subtasks. The scheduling algorithm is essentially the same as for the uniprocessor case without the 0/1 constraint. The only difference is that the edge capacities in the graph are somewhat different, and McNaughton's rule [McNaughton 59] is used to find the multiprocessor schedule in the final step.

4.2.1.2 Min-cost-max-flow Algorithms

Weighted task sets without the 0/1 constraint can be scheduled with minimum total error on both uniprocessor and multiprocessor systems using a slightly more complicated approach. As with the max-flow algorithm, uniprocessor scheduling can be performed for dependent task sets while multiprocessor scheduling requires independent task sets. In both cases the algorithm runs in time $O(n^6)$.

The algorithm works by constructing a weighted flow digraph. This is a flow digraph, where each edge e_i has a flow capacity c_i and a cost w_i. The cost of the flow through edge e_i is then $w_i f_i$, where f_i is the actual flow through e_i. A legal flow, of course, satisfies $f_i \leq c_i$. The graph will have a maximum legal flow $F = \sum_{e_i \in A'} f_i$, where A' are all edges out of the source vertex. The algorithm then finds the flow allotment that attains the maximum total flow F and minimizes the cost $C = \sum_{e_i \in A} c_i f_i$. Complete details of the digraph representation of the scheduling problem, with the network flow algorithm are given in [Shih 89a].

4.2.2 The Modified Earliest Deadline, and Other Fast Algorithms

Shi, Liu, and Chung [Shih 89b] present fast algorithms that can be used to preemptively schedule unweighted task sets with or without the 0/1 constraint. Dependent task sets without the 0/1 constraint can be scheduled on uniprocessor systems, while multiprocessor scheduling requires that the tasks be independent. Task sets with the 0/1 constraint can only be scheduled on uniprocessor systems, and all optional subtasks are required to have equal running times.

The first two algorithms are based upon the classical Earliest Deadline algorithm. In this algorithm the priorities are assigned based on the earliest approaching deadline of all ready tasks, and higher priority tasks that become ready may preempt a lower priority task that is running. Additionally, no tasks are allowed to compute past their deadlines, and are terminated even if they have not run to completion. The Earliest Deadline algorithm is modified to schedule task sets without the 0/1 constraint on both uniprocessors and multiprocessors with minimal total error. The uniprocessor scheduler finds schedules for unweighted dependent task sets. The uniprocessor algorithm can be modified using McNaughton's rule to schedule unweighted independent task sets on systems of identical processors. The uniprocessor algorithm runs in time $O(n \log n)$, while the multiprocessor algorithm runs in time $O(pn + n \log n)$ where p is the number of processors.

The authors [Shih 89b] also present algorithms to schedule uniprocessor task sets with the 0/1 constraint. Their Depth First Search algorithm can schedule unweighted dependent

task sets with the 0/1 constraint on uniprocessor systems when all the optional subtasks have equal running time in time $O(n^2)$. If the ready times of all the tasks are the same, a simpler Latest Deadline First algorithm can optimally schedule tasks on a uniprocessor system in time $O(n \log n)$.

5. APPLICATIONS

Trading time for accuracy can be very beneficial for many real-time applications where an imprecise but timely answer is more important than a late answer. The key to building such systems is to find ways to structure problems so that successive refinement algorithms are possible.

In this section we introduce several general algorithmic techniques that can be applied to soft-accuracy approaches, as well as a specific application involving real-time distributed databases. The techniques presented are not the classical paradigms of greedy, divide and conquer, or dynamic programming, but rather algorithmic techniques that lend themselves to soft-accuracy applications. The techniques are iterative partitioning and accumulation algorithms, and probabilistic algorithms including Monte Carlo approximation algorithms, Monte Carlo decision algorithms, and Las Vegas algorithms. Algorithms utilizing these techniques will have the appropriate error behavior required for soft accuracy approaches.

5.1 ITERATIVE BOUNDING ALGORITHMS

Iterative bounding algorithms converge to a solution by successive refinement of upper and lower bounds. The convergence may alternate between upper and lower bounds, as in a binary search algorithm, or it may proceed from a single bound. Algorithms that converge by squeezing a solution between two bounds are referred to as partitioning algorithms, while algorithms that converge from one direction only are referred to as accumulation algorithms. These algorithms can be contrasted with other iterative algorithms that do not make apparent progress towards a solution with each iteration. For example, the error in a well-behaved Taylor-Maclaurin series decreases monotonically as terms are added, and a computation terminated early may be clearly useful. For a dynamic programming calculation of Catalan numbers, it is not clear how to use the results of a prematurely terminated calculation.

Partitioning algorithms squeeze a solution between steadily approaching upper and lower bounds. Examples of partitioning algorithms include sectioning algorithms such as Newton's method for finding the root of an expression, branch and bound techniques, well-behaved series expansions with alternating terms, binary and interpolation searches, and other searching techniques.

Accumulation algorithms are similar to iterative bounding algorithms in that they converge by successive accumulation of the solution. The difference is that one of the bounds is not well defined. The algorithm then works by accumulating from the defined bound. Many greedy algorithms fall into this category since they build solutions by accumulation and do not reverse previous computations. Examples of these types of algorithms include minimal spanning tree algorithms, maximal graph matching by alternating path methods, minimal path algorithms, and series expansions with non-negative terms.

Iterative bounding algorithms are well suited for the imprecise computation method. The successive refinement nature of these algorithms makes them ideal candidates for use as optional tasks. The mandatory tasks can be computed either using the iterative algorithm with a fixed number of iterations, or using some other algorithm. Moreover, it is often possible to prove that the function of error with time has a particular shape (e.g., linearly decreasing, convexly decreasing). This information can be used in selecting appropriate scheduling algorithms [Chung 90].

5.2 PROBABILISTIC ALGORITHMS

Now let us move into the domain of probabilistic algorithms. We consider three categories of probabilistic algorithms: Monte Carlo approximation, Monte Carlo decision, and Las Vegas algorithms. For each category, we describe the type of stochastic relationship that exists between expended computation time and solution accuracy. We also suggest how the relationship might be used as the basis for making tradeoffs between timeliness and accuracy.

5.2.1 Monte Carlo Approximation Algorithms

Monte Carlo approximation algorithms were among the first probabilistic algorithms to be developed. These algorithms use randomness to approximate a solution, and have expected accuracy that increases with computation time. That is, approximations are produced on each iteration, and tend to get closer to the exact solution with increasing execution time.

As a simple example of such an algorithm consider a Monte Carlo approximation of π. In this algorithm there is a square region with side of length $2r$, and thus area $4r^2$. Inscribed in this square is a circle of radius r, with area πr^2. We now throw darts at this region so that they land randomly. The expected proportion of the total darts thrown that land in the inscribed circle is the ratio of the areas, $\frac{\pi r^2}{4r^2}$, which simplifies to $\frac{\pi}{4}$. Thus, if we throw a few darts and note the proportion landing inside the inscribed circle, we can expect to get

a rough approximation of π. If we throw many darts we can expect to get a more accurate approximation. Monte Carlo approximation algorithms have also been used for computing definite integrals, (the above example is actually a computation of the area of a circle), for determining queue length in complex queueing systems that have no closed form solution, and other numerical applications.

The time/accuracy characteristics of Monte Carlo approximation algorithms make them reasonable candidates for the imprecise computation method. At design time, the relationship between time and accuracy can be used to "size" the system. That is, enough computation time can be dedicated to the algorithms so that "acceptable" results can always be obtained. At run time, advantage can be taken of the relationship between time and accuracy; if extra processing time becomes available, it can be put to use effectively. That is, any unused processing time can be used to enhance the accuracy of the results.

Unfortunately, the convergence behavior often cannot be proved, so these algorithms require schedulers that do not depend upon any specific error behavior functions.

5.2.2 Monte Carlo Decision Algorithms

Monte Carlo decision algorithms can be distinguished from general Monte Carlo approximation algorithms in that they seek to arrive at a decision, not an increasingly precise approximation. Like the approximations, the decisions have a *probability of correctness* associated with them.

Monte Carlo decision algorithms always output a solution and have stable execution times, but make mistakes with probability p. The mistakes are often undetectable, but as with most probabilistic algorithms, the probability of an incorrect computation can be diminished arbitrarily by repeating the algorithm several times.

As an example consider the well-known Solovay-Strassen algorithm [Solovay and Strassen 77] for primality testing. In this algorithm, a witness is sought that probabilistically confirms the primality of the input number, and this witness can be found in polynomial time. All prime numbers, as well as some very small percentage of composite numbers, have such primality witnesses. The appearance of a primality witness for a composite number is referred to as a false witness since it falsely proclaims the composite number to be prime. When a primality witness is found the number can be assumed to be prime with high probability, but since some composite numbers have false witnesses, there is some small chance that the output is in error. Thus the algorithm quickly determines the primality of the number and errs with some fixed probability p.

The applicability of Monte Carlo decision algorithms to soft-accuracy approaches lies in our ability to calculate the probability of an incorrect computation. This probability of error can be diminished by repeated computations, which allows us to tailor the accuracy of the solution to the demands of the application. At design time, the algorithms can be allocated sufficient processor time to ensure acceptable probabilities of correct solutions. Once again, if additional processor time becomes available at run time, it can be utilized to produce a solution with a higher probability of correctness.

As an example of how "incorrect" solutions might be tolerable in a real-time system, suppose that a Monte Carlo decision algorithm is being used to distinguish targets from decoys. If a few decoys slip through the system as targets, then some weapons may be wasted, but catastrophic failure does not become imminent.

5.2.3 Las Vegas Algorithms

Las Vegas algorithms never return an incorrect answer, but the amount of time they take to arrive at a solution may be unbounded. That is, they sometimes never arrive at a solution, although the probability of arriving at a solution increases monotonically with computation time.

A simple example of this type of algorithm is choosing a leader on a token ring network. In this problem, n processors are tied to a communication network in the form of a one-way ring so that messages take exactly $n-1$ time units to reach all processors. The problem is for the processors to unanimously decide upon a leader. It should be noted that this problem has no deterministic solution if all processors run the same algorithm and start in the same initial state.

The Las Vegas solution runs in rounds. In the initial round all processors are *in*. In each successive round the *in* processors draw lots and the losers become *out*. Eventually only the leader remains *in* and the algorithm terminates. A round consists of two phases, a roulette phase and an evaluation phase. In the roulette phase all *in* processors randomly choose between the bits 0 and 1. All processors that chose 0 are then *out*, and all processors that chose 1 remain *in*. In the evaluation phase, all *in* processors put a message out on the ring and wait. If no messages appear on the ring (it takes exactly $n-1$ time units for each processor to determine this), then all the *out* processors from that round are reinstated and that round is repeated. The leader is chosen when a round is reached with exactly one *in* processor remaining, and hence one message on the ring.

Las Vegas algorithms can be used as primary tasks in the backup approximation method. If the algorithm is successful, a precise solution is produced before the alternate task must

run. The probability distribution for attaining a solution with time can generally be calculated for Las Vegas algorithms. This predictability makes it possible for tradeoffs between time and accuracy to be made at both design time and run time. At design time, enough time can be dedicated to a Las Vegas algorithm so that the probability of attaining a solution is acceptable. If more time can be allocated to the algorithm at run time, the probability of attaining a solution can be increased.

5.3 PARTIAL QUERY COMPUTATIONS

Soft-accuracy approaches are useful for distributed databases that must operate in real-time situations, where communications delays, failing nodes, and wide distribution of data make it difficult to meet hard deadlines. Traditional databases work in an all-or-nothing manner. That is, a query cannot be answered unless all information is available. Davidson and Watters [Davidson and Watters 88], and Smith and Liu [Smith and Liu 89] outline techniques that yield inexact solutions to relational queries given partial information. A refinement of the technique using an object-oriented approach is found in [Vrbsky 89].

The systems yield a series of approximations $R_1, R_2, \ldots$ that converge to the exact solution R. Each approximation in the series is a refinement of the previous approximation computed from previously unavailable information. The approximations R_i represent complete and consistent approximations of R. They are complete in that no tuple in R is left out of any approximation R_i, and consistent in that it is possible to compute from any R_i tuples that are not in R. The approximation is exact when no membership uncertainty exists for any tuple.

6. SUMMARY

This paper introduces the concept of soft accuracy as an approach for addressing hard real-time scheduling without resorting to excessive overbuild. Soft-accuracy is presented as a natural extension of the "softening" concept used in other approaches such as softened utilization under rate monotonic scheduling, and soft-deadline approaches. The soft-accuracy approach allows both robust overload performance and lean hardware requirements. It is argued that these characteristics are well suited for SDS applications. Two methods utilizing soft accuracy are presented, and examples of application areas where soft accuracy is applicable are given.

Of the two methods presented, the imprecise computation scheduling algorithms provide the most depth. The problems of scheduling both periodic and general deadline driven systems on uniprocessor and multiprocessor systems are explored. Periodic systems are

further refined by the cumulative or non-cumulative effect that errors have on the system. Algorithms are presented that run in times ranging from $O(n \log n)$ to $O(n^6)$ for a variety of scheduling situations. Finally, examples of application areas amenable to the soft-accuracy approach are presented. These include problems solvable by accumulation algorithms, partitioning algorithms, and probabilistic algorithms, and Partial Query evaluations in relational databases.

References

[Blazewicz and Finke 87] Blazewicz, J. and G. Finke, "Minimizing Mean Weighted Execution Time Loss on Identical and Uniform Processors," *Information Processing Letters 24*, March 1987, 259-263.

[Chung 90] Chung, J. Y., J. W. S. Liu, and K. J. Lin, "Scheduling Periodic Jobs that Allow Imprecise Results," to appear in *IEEE Transactions on Computers.*

[Davidson and Watters 89] Davidson, S. B. and A. Watters, "Partial Computation in Real-Time Database Systems," *Proceedings of the Fifth IEEE Workshop on Real-Time Operating Systems and Software*, Washington, D.C., May 12-13, 1988, 117-121.

[Ford and Fulkerson 62] Ford, Jr., L. R. and D. R. Fulkerson, *Flows in Networks*, Princeton University Press, 1962.

[Gear 88] Gear, C. W., K. J. Lin, C. L. Liu, and J. W. S. Liu, "Algorithmic Aspects of Real-Time Computing," *Kickoff Workshop of ONR Foundations of Real-Time Computing Research Initiative*, Falls Church, VA, November 1988, 17-22.

[Johnson 73] Johnson, D. S., *Near Optimal Bin Packing Algorithms*, Technical Report MAC TR-109, Project MAC, MIT, Cambridge, MA, 1973.

[Karp and Rabin 87] Karp, R. M. and M. O. Rabin. "Efficient Randomized Pattern-Matching Algorithms," *IBM Journal of Research and Development 31*, 2 (March 1987).

[Lawler and Moore 69] Lawler, E. L. and J. M. Moore, "A Functional Equation and its Applications to Resource Allocation and Scheduling Problems," *Management Science 16*, 1969, 77-84.

[Leung 89] Leung, J. Y. T., "A New Algorithm for Periodic Real-Time Tasks," *Algorithmica 4*, 1989, 209-219.

[Liestman and Campbell 80] Liestman, A. L. and R. H. Campbell, *A Fault-Tolerant Scheduling Problem*, Report No. UIUCDCS-R-80-1010, Dept. of Computer Science, University of Illinois, Urbana, IL, 1980.

[Liestman and Campbell 86] Liestman, A. L. and R. H. Campbell, "A Fault-Tolerant Scheduling Problem," *IEEE Transactions on Software Engineering SE-12*, 11 (November 1986), 1089-1095.

[Liu 89] Liu, J. W. S., K. J. Lin, C. L. Liu, and C. W. Gear, "Research on Imprecise Computations in Project QuartZ," *1989 Workshop on Operating Systems for Mission Critical Computing*, College Park, MD, September 1989.

[Liu and Layland 73] Liu, C. L. and J. W. Layland, "Scheduling Algorithms for Multiprocessing in a Hard Real-Time Environment," *Journal of the ACM 20*, 1 (January 1973), 46-61.

[McNaughton 59] McNaughton, R., "Scheduling With Deadlines and Loss Functions," *Management Science 12*, 1959, 1-12.

[Shih 89a] Shih, W. K., J. W. S. Liu, J. Y. Chung, and D. W. Gillies, "Scheduling Tasks with Ready Times and Deadlines to Minimize Average Error," *ACM Operating Systems Review 23*, 3 (July 1989), 14-28.

[Shih 89b] Shih, W. K., J. W. S. Liu, and J. Y. Chung, *Fast Algorithms for Scheduling Imprecise Computations with Timing Constraints*, Report No. UIUCDCS-R-89-1506, Dept. of Computer Science, University of Illinois, Urbana, IL, May 1989.

[Smith and Liu 89] Smith, K. P. and J. W. S. Liu, "Monotonically Improving Approximate Answers to Relational Algebra Queries," *Proceedings of IEEE COMPSAC*, Orlando, FL, September 1989.

[Solovay and Strassen 77] Solovay, R. and V. Strassen, "A Fast Monte-Carlo Test for Primality," *SIAM Journal of Computing 6*, 1 (March 1977), 84-85.

[Vrbsky 89] Vrbsky, S. V., J. W. S. Liu, and K. P. Smith, "An Object Oriented Data Model for Monotone Approximate Query Processing," Department of Computer Science, University of Illinois, Urbana, IL, 1989.

APPENDIX D

SCHEDULING APERIODIC TASKS WITH HARD DEADLINES IN A RATE MONOTONIC FRAMEWORK[1]

ABSTRACT

It has been suggested [Sha 86] that an aperiodic task with hard deadlines can be accommodated by the rate monotonic scheduling algorithm in the following way: first, assume some minimal separation time between arrivals of the task; then, treat the task as if it were a periodic task with a period equal to the minimal separation time.

Two phenomena must be considered when an aperiodic task is handled in this way: multi-arrival periods and interperiod gaps. Multi-arrival periods are periods in which two or more aperiodic arrivals occur. They arise whenever *actual* interarrival times can be less than the *assumed* minimal separation time. In a multi-arrival period, all arrivals except the one that initiates the period are ignored, or rejected, by the rate monotonic scheduling algorithm. Interperiod gaps are the time intervals between the end of one period and the subsequent arrival of the aperiodic task, which initiates the next period. During these gaps, the rate monotonic algorithm has reserved processor capacity for the aperiodic task, but none is used. Thus, interperiod gaps can lead to low average processor utilization, or system "overbuild."

In this paper, we analyze these two phenomena for the case in which the aperiodic task has a Poisson arrival process. We derive analytic expressions for: (1) the probability of a period experiencing multiple arrivals, as a function of the assumed minimal separation time, and (2) the percentage of time represented by interperiod gaps, also as a function of the minimal separation time. These results quantify the degree to which multi-arrival periods become *more* of a problem and interperiod gaps become *less* of a problem as the assumed minimal separation time increases.

1. RATE MONOTONIC SCHEDULING ALGORITHM

Let us consider a set of m periodic tasks, $\tau_1, \ldots, \tau_m$, with task τ_i having periods (or constant interarrival times) of T_i and computation requirements of C_i, $i = 1, \ldots, m$. Each arrival of task τ_i initiates a period of length T_i time units, during which it must receive C_i time units of computation. The end of the period represents the "hard deadline" of the arrival that initiated the period. It also coincides with the occurrence of the next arrival, which marks the beginning of the next period. Since task τ_i requires C_i time units of computation every T_i time units, its utilization of the processor is C_i/T_i. [2]

---

1. This paper appeared in *Proceedings of the Sixth IEEE Workshop on Real-Time Operating Systems and Software*, May 1989, 1-5. The co-authors were Karen D. Gordon of IDA, Lawrence W. Dowdy of Vanderbilt University, James Baldo, Jr., of IDA, and Kevin J. Rappoport of IDA.

2. Our notation is based on the notation in [Liu and Layland 73].

The rate monotonic algorithm is an optimal preemptive, static-priority-driven uniprocessor scheduling algorithm for periodic tasks with hard deadlines as defined above.[3] As a *preemptive, priority-driven* algorithm, it ensures that the processor is always executing the highest priority task in the processor ready queue (unless of course the queue is empty, in which case the processor is idle). If a task of higher priority than the currently executing task joins the queue, then the currently executing task is preempted, and the processor begins executing the newly arrived (higher priority) task.

As a *static* algorithm, the rate monotonic algorithm assigns priorities that are determined prior to execution and remain fixed over time. In particular, it assigns (static) priorities to tasks according to the lengths of their periods: tasks with shorter periods are assigned higher priorities. Alternatively stated, tasks with higher arrival rates are assigned higher priorities, making task priority a monotonically increasing function of task arrival rate; hence the term "rate monotonic."

The rate monotonic algorithm was shown to be optimal within the class of preemptive, static-priority-driven scheduling algorithms by Liu and Layland [Liu and Layland 73]. This means that no other algorithm of this class can schedule a set of periodic tasks that cannot also be scheduled by the rate monotonic algorithm. In addition to establishing optimality, Liu and Layland established a *sufficient* condition for the schedulability of an arbitrary task set. The condition is that the total processor utilization U of the task set is no more than $ln\ 2$, i.e.,

$$U = \sum_{i=1}^{i=m} (C_i/T_i) \leq ln\ 2\ (\approx 0.693)$$

Thus, for any task set of any size whose total processor utilization is less than or equal to $ln\ 2$, the rate monotonic algorithm guarantees that all deadlines of the task set will be met. Under more restrictive conditions, the total utilization can be higher.

The rate monotonic algorithm is appealing because of its low overhead, efficiency, predictability, and extensibility. It is *low-overhead* in the sense that priority assignments are static; in other words, it is low-overhead relative to dynamic algorithms. It is *efficient* in two senses. First, it is optimal with respect to the class of static algorithms. Second, it is viewed as being competitive with respect to the class of dynamic algorithms. That is, the "overbuild" required to achieve a processor utilization of no more than $ln\ 2$ is generally regarded as being "acceptable." Finally, it is *predictable* in the sense that (under the specified conditions) deadlines are guaranteed to be met *a priori*. The need for exhaustive testing is eliminated.

---

3. The earliest deadline and least slack time algorithms are optimal preemptive, *dynamic*-priority-driven scheduling algorithms [Liu and Layland 73] [Mok 83].

Perhaps the most important feature of the rate monotonic algorithm is its "extensibility." It has proven to be amenable to a number of extensions beyond traditional periodic task scheduling. For example, extensions have been developed for dealing with transient overload [Sha 87], task synchronization [Rajkumar 88] [Sha 88], and aperiodic tasks.

In regard to aperiodic tasks, several algorithms for accommodating aperiodic tasks *without hard deadlines* in a rate monotonic framework have been developed [Lehoczky 87] [Sha 87] [Sprung 88]. These algorithms seek to reduce average response times of aperiodic tasks while still guaranteeing hard deadlines of periodic tasks.

2. EXTENSION FOR ACCOMMODATING APERIODIC TASKS WITH HARD DEADLINES

Now let us focus on aperiodic tasks *with hard deadlines*. The following approach has been proposed for accommodating this class of tasks in a rate monotonic framework [Sha 86]:

a. For aperiodic task τ_k, assume some minimal separation time L_k between arrivals of the task.

b. Treat task τ_k as if it were periodic with a period of length L_k. That is, assign task τ_k a priority based on L_k, according to the rate monotonic priority assignment algorithm.

Clearly, the minimal separation time L_k is a critical parameter in this approach. In some cases, physical conditions may dictate a value for L_k, but, in general, the choice of a value for L_k is not so straightforward. For example, if the aperiodic task has a Poisson arrival process, then arrivals can occur at arbitrarily close time instants. For any chosen value of L_k (> 0), there is some probability of encountering interarrival times of a still lower value. On one hand, the value of L_k must be low enough to make this probability tolerable. On the other hand, the value of L_k must not be too low, or an intolerable level of excess processor capacity will be dedicated to τ_k.

2.1 DISCUSSION

The interarrival times of an aperiodic task are, by definition, variable. This variability leads to two phenomena: multi-arrival periods and interperiod gaps. The choice of L_k determines the relative prevalence of these two phenomena. As suggested above, both have undesirable consequences, which threaten certain key features of the rate monotonic scheduling algorithm, namely predictability and efficiency. These consequences are explained below.

2.1.1 Multi-Arrival Periods

Let us consider the impact of multi-arrival periods. Given that task τ_k has computation requirement C_k, the rate monotonic scheduling algorithm, in effect, reserves a processor utilization of $U_k = C_k/L_k$ for task τ_k. To maintain predictability in the form of guaranteed deadlines for all tasks in the task set, task τ_k must not exceed this allocated utilization. Thus, one and only one arrival can be scheduled in any given period. If multiple arrivals do occur within a period, all except the one that initiates the period are rejected. Clearly, rejected arrivals miss their deadlines, since they receive no service at all. These rejected arrivals should be taken into account during the process of selecting a value for L_k.

2.1.2 Interperiod Gaps

Now, consider the impact of interperiod gaps. The beginning of a period no longer coincides with the ending of the preceding period, but instead occurs some time later. Thus, periods of task τ_k become separated by intervals of time during which the next arrival of task τ_k is being awaited. These intervals represent "idle" time, with respect to task τ_k; the rate monotonic algorithm has reserved processor capacity (of C_k/L_k) for τ_k, but none is used. This dedicated but unused processor capacity should also be taken into account during the process of selecting a value for L_k.

2.2 ANALYSIS

Let us assume that task τ has a Poisson arrival process with arrival rate λ.[4] Then, arrivals of task τ have exponentially distributed interarrival times with a mean of $1/\lambda$. That is, if T is a random variable representing interarrival time, the cumulative distribution function of T is

$$F_{\mathrm{T}}(t) = P(\mathrm{T} \leq t) = 1 - e^{-\lambda t}$$

The following results are derived on the basis of this assumption.[5]

2.2.1 Multi-Arrival Periods

For task τ with Poisson arrival process of rate λ, the probability of a period having multiple arrivals is simply the probability that the arrival following the initial one occurs prior to the lapse of the minimal separation time L. Denote this probability by $M_{\lambda, L}$. Then, from the formula for the cumulative distribution of the exponential distribution,[6]

4. For the sake of clarity, we now drop the subscript indicating task number.

5. The reader is referred to [Kleinrock 75] for a review of the Poisson process and the exponential distribution.

$$M_{\lambda, L} = P(\mathrm{T} < L) = 1 - e^{-\lambda L}$$

Suppose that we wish to achieve a specified value of $M_{\lambda, L}$, say M_0 (e.g., 10%, 1%, etc.). We can do so by solving the above equation for L, setting $M_{\lambda, L}$ to M_0, and assigning the resulting value to L as follows:

$$L = -\frac{1}{\lambda} ln(1 - M_0)$$

2.2.2 Interperiod Gaps

Here, we are interested in the fraction of time represented by interperiod gaps. Denote this fraction by $I_{\lambda, L}$. If the mean length of interperiod gaps is denoted by $G_{\lambda, L}$, then $I_{\lambda, L}$ can be written as follows:

$$I_{\lambda, L} = \frac{G_{\lambda, L}}{G_{\lambda, L} + L}$$

Now, the problem is to derive $G_{\lambda, L}$.

In other words, we want to derive the mean time from the end of a period to the next arrival of the task. By virtue of the memoryless property of the exponential distribution [Kleinrock 75, p. 66], the mean time from the end of a period to the next arrival is independent of how much time has passed since the previous arrival. So, the mean time from the end of a period to the next arrival is simply equal to the mean interarrival time $1/\lambda$. That is,

$$G_{\lambda, L} = \frac{1}{\lambda}$$

Therefore, $I_{\lambda, L}$ is given by the following equation:

$$I_{\lambda, L} = \frac{\frac{1}{\lambda}}{\frac{1}{\lambda} + L}$$

Again, we can solve for L to obtain the value of L necessary to achieve a specified level of $I_{\lambda, L}$, say I_0, as follows:

$$L = \frac{\frac{1}{\lambda}(1 - I_0)}{I_0}$$

---

6. Note that since T is a continuous random variable, $P(\mathrm{T} < L) = P(\mathrm{T} \leq L)$.

Finally, we can calculate the "excess" processor capacity $E_{\lambda, L}$ dedicated to task τ. We do so by recognizing that during interperiod gaps, the rate monotonic algorithm still reserves a processor utilization of C/L for task τ. We obtain the following result:

$$E_{\lambda, L} = (C/L)\, I_{\lambda, L}$$

$E_{\lambda, L}$ represents the cost, in terms of efficiency, of maintaining predictability, in the form of guaranteed deadlines.

In Figure 1, both $M_{\lambda, L}$ and $I_{\lambda, L}$ are plotted as functions of L.

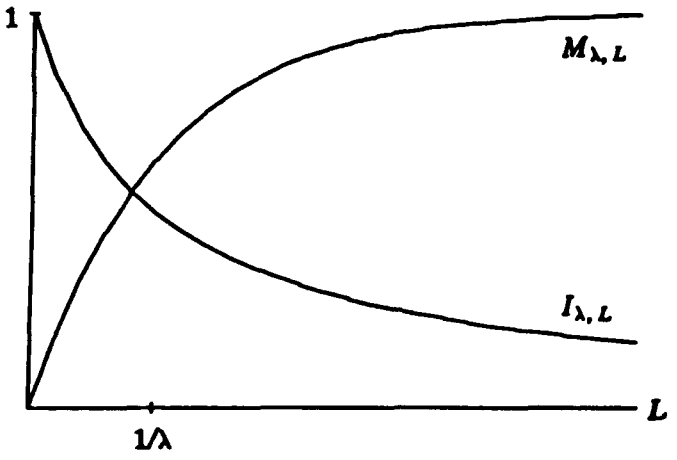


Figure 1. Tradeoff Between Multi-Arrival Periods and Interperiod Gaps

3. CONCLUDING REMARKS

As shown Figure 1, "desirable" (namely, low) values of $M_{\lambda, L}$ (the probability of a period having multiple arrivals) occur at *low* values of the assumed minimal separation time L, whereas "desirable" (again, low) values of $I_{\lambda, L}$ (the fraction of time represented by interperiod gaps) occur at *high* values of L.[7] Herein lies the tradeoff that must be considered when selecting a value for the minimal separation time L. If tolerable values of both $M_{\lambda, L}$ and $I_{\lambda, L}$ cannot be attained at the same time (i.e., at the same value of L), then an alternative scheduling approach must be considered.

7. The intersection of the curves lies at approximately 0.8 ($1/\lambda$).

REFERENCES

[Kleinrock 75] Kleinrock, L., *Queueing Systems, Volume 1: Theory*, John Wiley & Sons, Inc., 1975.

[Lehoczky 87] Lehoczky, J.P., L. Sha, and J.K. Strosnider, "Enhanced Aperiodic Responsiveness in Hard Real-Time Environments," *IEEE Real-Time Systems Symposium*, December 1987.

[Liu and Layland 73] Liu, C.L. and J.W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment," *Journal of the ACM 20*, 1 (January 1973), 46-61.

[Mok 83] Mok, A.K., *Fundamental Design Problems of Distributed Systems for the Hard Real-Time Environment*, Ph.D. Thesis, Department of Electrical Engineering and Computer Science, M.I.T., 1983.

[Rajkumar 88] Rajkumar, R., L. Sha, and J.P. Lehoczky, "Real-Time Synchronization Protocols for Multiprocessors," *IEEE Real-Time Systems Symposium*, December 1988.

[Sha 86] Sha, L., J.P. Lehoczky, R. Rajkumar, "Solutions for Some Practical Practical Problems in Prioritized Preemptive Scheduling," *IEEE Real-Time Systems Symposium*, December 1986.

[Sha 87] Sha, L., J.P. Lehoczky, and R. Rajkumar, "Task Scheduling In Distributed Real-Time Systems," *Proceedings of IEEE Industrial Electronics Conference*, 1987.

[Sha 88] Sha, L., R. Rajkumar, and J.P. Lehoczky, "Priority Inheritance Protocols: An Approach to Real-Time Synchronization," Departments of CS, ECE, and Statistics, Carnegie Mellon University, 23 May 1988.

[Sprung 88] Sprung, B., J.P. Lehoczky, and L. Sha, "Exploiting Unused Periodic Time for Aperiodic Service Using the Extended Priority Exchange Algorithm," *IEEE Real-Time Systems Symposium*, December 1988.

Distribution List for IDA Paper P-2422

NAME AND ADDRESS	NUMBER OF COPIES
Sponsor	
Lt Col James Sweeder Chief, Computer Resources Engineering Division SDIO/ENA The Pentagon, Room 1E149 Washington. D.C. 20301-7100	2
Other	
Dr. Ashok Agrawala Department of Computer Science University of Maryland College Park, MD 20742	1
Dr. Jon Agre Science Center Rockwell International Corporation Mail Stop A24 1049 Camino Dos Rios Thousand Oaks, CA 91360	1
Dr. Dan Alpert, Director Program in Science, Technology & Society University of Illinois Room 201 912-1/2 West Illinois Street Urbana, IL 61801	1
Mr. Rich Bergman Naval Ocean Systems Center San Diego, CA 92152-5000	1
Dr. Barry W. Boehm Director, DARPA/SISTO 3701 North Fairfax Drive Arlington, VA 22203-1714	1
LtCol Brian Boesch DARPA/CSTO 3701 North Fairfax Drive Arlington, VA 22203-1714	1

NAME AND ADDRESS	NUMBER OF COPIES
Mr. Dale Brouhard Naval Ocean Systems Center San Diego, CA 92152-5000	1
Ms. Virginia L. Castor Special Assistant for Software and Computer Technology ODDDRE(R&AT) Room 3E114, Pentagon Washington, DC 20301-3080	1
Dr. David R. Cheriton Computer Science Department Bldg. 460, Room 422 Stanford University Stanford, CA 94305-6110	1
Mr. William M. Corwin Intel Corp. HF3-64 5200 NE Elam Young Parkway Hillsboro, OR 97124	1
Defense Technical Information Center Cameron Station Alexandria, VA 22314	2
Dr. Larry Dowdy Computer Science Department Vanderbilt University P.O. Box 1679, Station B Nashville, TN 37235	1
Mr. Walt Heimerdinger Software Engineering Institute Carnegie Mellon University Pittsburgh, PA 15213	1
Ms. Connie Heitmeyer Code 5534 Naval Research Lab 4555 Overlook Avenue, SW Washington, DC 20375	1
Mr. Steve Howell NSWC - White Oak 10901 New Hampshire Avenue Silver Spring, MD 20903-5000	1

NAME AND ADDRESS	NUMBER OF COPIES
Mr. Phil Hwang NSWC - White Oak Code U-33 10901 New Hampshire Avenue Silver Spring, MD 20903-5000	1
Mr. Richard Iliff SDIO ENA Room 1E149, The Pentagon Washington, D.C. 20301	1
Mr. D.P. Juttelstad Naval Underwater Systems Center Newport Laboratory Newport, RI 02841	1
Dr. Virginia P. Kobler US Army Strategic Defense Command P.O. Box 1500 Huntsville, AL 35807-3801	1
Dr. Gary M. Koob Computer Science Division, Code 1133 Office of Naval Research 800 N. Quincy Street Arlington, VA 22217-5000	1
Dr. John F. Kramer STARS Technology Center 3701 North Fairfax Drive Arlington, VA 22204-1714	1
Mr. Tom Lawrence Rome Laboratory RL/COAC Griffis AFB, NY 13441-5700	1
Dr. John P. Lehoczky Department of Statistics Carnegie Mellon University Pittsburgh, PA 15213-3890	1
Dr. Charles Lillie SAIC 1710 Goodridge Drive P.O. Box 1303 McLean, VA 22102	1

NAME AND ADDRESS	NUMBER OF COPIES
Mr. J. Oblinger Naval Underwater Systems Center Newport Laboratory Newport, RI 02841	1
Mr. Frank Prindle Naval Air Development Center Warminster, PA 18974	1
LTC Mark Pullen DARPA/ASTO 3701 North Fairfax Drive Arlington, VA 22203-1714	1
Dr. Richard F. Rashid Department of Computer Science Carnegie Mellon University Pittsburgh, PA 15213-3890	1
Mr. Helmut Roth NSWC - White Oak 10901 New Hampshire Avenue Silver Spring, MD 20903-5000	1
CDR(Sel) Greg Sawyer SPAWAR 231-2B2 Space and Naval Warfare Systems Command Washington, DC 20363-5100	1
Dr. John Salasin Software Engineering Institute and School of Computer Science Carnegie Mellon University Pittsburgh, PA 15213-3890	1
Dr. William Scherlis DARPA/SISTO 3701 North Fairfax Drive Arlington, VA 22203-1714	1
SDIO Technical Information Center DRC 1755 Jeff Davis Highway Suite 802 Crystal Square 5 Arlington, VA 22202	1

NAME AND ADDRESS	NUMBER OF COPIES
Mr. J. Oblinger Naval Underwater Systems Center Newport Laboratory Newport, RI 02841	1
Mr. Frank Prindle Naval Air Development Center Warminster, PA 18974	1
LTC Mark Pullen DARPA/ASTO 3701 North Fairfax Drive Arlington, VA 22203-1714	1
Dr. Richard F. Rashid Department of Computer Science Carnegie Mellon University Pittsburgh, PA 15213-3890	1
Mr. Helmut Roth NSWC - White Oak 10901 New Hampshire Avenue Silver Spring, MD 20903-5000	1
CDR(Sel) Greg Sawyer SPAWAR 231-2B2 Space and Naval Warfare Systems Command Washington, DC 20363-5100	1
Dr. John Salasin Software Engineering Institute and School of Computer Science Carnegie Mellon University Pittsburgh, PA 15213-3890	1
Dr. William Scherlis DARPA/SISTO 3701 North Fairfax Drive Arlington, VA 22203-1714	1
SDIO Technical Information Center DRC 1755 Jeff Davis Highway Suite 802 Crystal Square 5 Arlington, VA 22202	1

NAME AND ADDRESS	NUMBER OF COPIES
LCDR Robert Voigt SPAWAR 231-2B1 Space and Naval Warfare Systems Command Washington, DC 20363-5100	1
Dr. Ralph Wachter ONR (Code 1133) 800 N. Quincy Street Arlington, VA 22217-5000	1
Ms. Elizabeth Wald Code 5150 Naval Research Laboratory 4555 Overlook Avenue, SW Washington, DC 20375-5000	1
Mr. Paul Wallenberger NSWC - White Oak 10901 New Hampshire Avenue Silver Spring, MD 20903-5000	1
IDA	
General Larry D. Welch, HQ	1
Mr. Philip L. Major, HQ	1
Dr. Robert E. Roberts, HQ	1
Ms. Ruth L. Greenstein, HQ	1
Mr. James Baldo, CSED	1
Ms. Anne Douville, CSED	1
Mr. Stephen Edwards	1
Dr. Dennis W. Fife, CSED	2
Dr. Karen D. Gordon, CSED	30
Dr. Richard J. Ivanetich, CSED	1
Mr. Terry Mayfield, CSED	1
Dr. Reginald N. Meeson, CSED	1
Ms. Katydean Price, CSED	2
Ms. Beth Springsteen, CSED	1
Dr. Richard L. Wexelblat, CSED	1
Mr. David Wheeler, CSED	1
Mr. Kevin J. Rappoport, SRC	5
IDA Control & Distribution Vault	3